Distributed Active XML and Service Interfaces

Loïc Hélouët, Albert Benveniste

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Distributed Active XML and Service Interfaces

Loïc Hélouët, Albert Benveniste

N° 7082

October 2009

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--7082--FR+ENG

Distributed Active XML and Service Interfaces*

Loïc Hélouët, Albert Benveniste

Thème COM — Systèmes communicants
Équipes-Projets Distribcom

Rapport de recherche n° 7082 — October 2009 — 61 pages

Résumé : Guarded Active XML (GAXML) a été proposé par Abiteboul, Segoufin, and Vianu comme langage de spécification pour des web-services dynamiques et orientés données. GAXML consiste en un ensemble de documents XML dans lesquels sont intégrés des appels de services gardés. Cette extension de XML permet de définir des flots de controle dans des documents structurés.

Ce rapport propose une extension de GAXML à l'aide de concepts nécessaires aux exigences des architectures orientées services. Nous proposons un modèle plus riche pour la définition des services externes à un site, sous la forme *d' interfaces*. Spécifier une interface consiste à décrire, à l'aide de patterns: 1/ l'aspect des documents qui seront utilisés comme paramètres d'un appel à un service externe, 2/ les resultats possibles renvoyés par le service externe. Notre notion d'interface est bien sur liée à celle d'implémentation —un service implémente une interface — qui s'appuie sur le concept bien étudié de *containment* de requêtes.

Nous proposons ensuite *Distributed Active AXML* (DAXML) comme modèle pour la distribution de services en AXML gardé sur une architecture composée d'un ensemble d'agents. Les agents transformed des documents AXML en réponse à des appels de service provenant de leur propre localité, ou envoyés de manière asynchrone par d'autres agents. Les systèmes DAXMLainsi construits se composent, en remplacant la description (par une interface) d'un services distant par un appel à une implémentation distante offerte par un autre agent. Les systèmes DAXMLse *raffinent* également par remplacement de services externes par une implémentation locale rendant le même service. Une opération symétrique *d'abstraction* est également possible. L'abstraction de services réels par des interfaces est un outil efficace pour simplifier l'analyse de systèmes DAXML. Nous illustrons ces concepts sur un exemple représentatif combinant données et flot de contrôle, la chaîne de production Dell.

Mots-clés : AXML, web services, Interfaces

* This work was partially funded by the ANR national research program DOTS (ANR-06-SETI-003), DocFlow (ANR-06-MDCA-005) and the project CREATE ActivDoc.

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex (France)
Téléphone : +33 2 99 84 71 00 — Télécopie : +33 2 99 84 71 71

Distributed Active XML and Service Interfaces

Abstract: Guarded Active XML (GAXML) was proposed by Abiteboul, Segoufin, and Vianu, as a high-level specification language tailored for data-intensive, distributed, dynamic Web services. GAXML consists in XML documents with embedded guarded service calls, thus allowing for the definition of control flows in documents. In this paper we enhance GAXML with the concepts needed to satisfy the requirements of "Service Computing" and "Service Oriented Architectures". We provide a richer model for external services in the form of *interfaces*. Specifying an interface consists in describing, using patterns: 1/ the shape of documents that can serve as parameters to a call, and 2/ the possible returns of a call. Our notion of interface comes with a notion of *implementation* — a service implements an interface — that builds upon the known concept of containment and a new concept of satisfaction. Then, we propose *Distributed Active AXML* (DAXML) as a model of guarded active XML systems distributed over a set of peers. Peers transform distributed documents in response to service calls from their own or other peers in an asynchronous way. DAXML systems *compose*, thus capturing the mechanism of replacing an external service call by a distant call to an implementation of it, offered by another peer. DAXML systems and documents can be *refined* by replacing, in documents, external service calls by respective implementations thereof; the symmetric operation is service *abstraction*. Abstracting services as interfaces is an efficient tool in simplifying analyses of DAXML documents. We illustrate our approach on a representative example combining data and workflow management, namely the Dell supply chain.

Key-words: AXML, web services, Interfaces

Table des matières

1 Introduction **4**

2 Background on Documents, Patterns, and Queries **6**
2.1 Trees, Forests, and Patterns 6
2.2 Queries 10

3 The Distributed Guarded AXML Model **12**
3.1 Services as Functions and Interfaces 13
3.1.1 Functions 13
3.1.2 Interfaces 15
3.2 Implementation 18
3.2.1 Containment 18
3.2.2 Satisfaction 19
3.2.3 Implementation of an interface 22
3.2.4 Discussion 23
3.3 DAXML Schemas and Instances 25
3.3.1 Schemas and their composition 25
3.3.2 Instances and their runs 27
3.3.3 Projections and restrictions 30
3.3.4 Refinement of services and instances 33
3.4 DAXML versus classical SOA 34

4 Analysis of DAXML systems **35**
4.1 Reachability 35
4.2 SOA modalities 36
4.3 Beyond static interfaces 36
4.4 Tree-LTL logic 37

5 A DAXML example : the Dell supply chain **42**
5.1 DAXML modeling of Dell supply chain : architecture and schemas . 43
5.2 DAXML description of DellOrder 45
5.3 Using service abstraction to simplify property checking 54

6 Conclusion **54**

A Appendix : proofs **57**
A.1 Proof of Theorem 1 57
A.2 Proof of Theorem 2 57
A.3 proof of Theorem 3 57
A.4 Proof of theorem 4 58
A.5 Proof of theorem 8 60
A.6 proof of corollary 2 60
A.7 Proof of corollary 3 61
A.8 Proof of theorem 12 61

1 Introduction

Service-oriented architectures (SOA) have become very popular since the 90's, and are seen as a way to speed up development processes, but also to build more adaptable software. The main concepts in SOA are services and interfaces. A service is a software unit that provides a set of operations. The way to use these operations is published as an interface. One of the major assumptions is that when a service is used as specified in its interface, it should act as promised (usually return a correct value). A Service Oriented Architecture consists in a collection of interacting services, that are composed. Several technical solutions have been proposed to implement SOA over dedicated middlewares (usually called a service bus) or over the web (the term *web services* is then used), but so far very few formal models exist to reason about data-driven distributed applications.

Active XML (AXML) was proposed by Abiteboul, Benjelloun, and Milo [1], as a high-level specification language tailored for data-intensive, distributed, dynamic Web services. Active XML mainly consists in XML documents [25] with embedded service calls. AXML offers mechanisms to store and query structured data distributed over entities called *peers*. A peer owns several services that can be called either locally or by other peers. One of the main original concepts of AXML is to allow *lazy evaluation*, that is services are allowed to return structured data that contain references to services that have to be called to continue the evaluation of the returned values if needed.

The concepts of Peers and services can be seen as offering the needed features to implement SOA architectures. However, in SOA, services do not only search data or compute results to answer a call, but may also orchestrate the execution of several services. However, in its initial version, AXML did not allow for the definition of control flows, which are a central need for the definition of business processes and workflow management. A first attempt to support workflow management in Active XML was proposed by Abiteboul, Segoufin, and Vianu [3], with Guarded Active XML (GAXML). Guarded AXML adds guards to services, which allows for the definition of control flows. In addition to this guarding mechanism, [3] studies decidability and complexity of model checking for Tree-LTL (a linear tree-pattern based temporal logic) on runs of Guarded AXML systems.

However, GAXML does not yet deal with the distribution, which is a central concept in SOA. In SOA, applications are composed of several services, that must work on any architecture, and communicate only through messages. Services know each other only through contracts (that describe the needs of a service that another service must fulfill) and interfaces (that describe the functionalities provided by a service). In Service Computing and Service Oriented Architectures, *services* are exposed through *interfaces* that abstract away implementation details. Interfaces specify how a service must be called and what it is expected to return. In rich interface theories [8, 14], interfaces can even be used to specify the composition of services. In its 2008 form, GAXML provides most of the features to allow AXML based SOAs, but still does not take distribution into account as GAXML guards apply to the whole modeled system, without taking into account the fact that data is distributed over a compartmentalized architecture.

In GAXML, the two notions of service and interface correspond to the concepts of *internal* and *external functions*, respectively. Internal functions describe some operations on documents, and external ones are simply a constraint on the outputs returned by an external implementation, expressed as the conjunction of a DTD and a combination of tree patterns. Distribution is only reflected by the localization of a service on a peer, and GAXML systems compose by summing up both systems, and replacing some external services by internal ones.

As a first contribution of this paper, we equip GAXML with a richer notion of *interface*, based on an enhanced notion of *pattern*. The patterns we use are finite sets of classical tree patterns [4] extended with variables and constraints over these variables. An interface consists of a *call pattern* and a finite set of *return patterns*. The call pattern describes what the input to any service implementing the given interface should look like. The return patterns describe the possible shape of the expected output provided by any service implementing this interface, with a disjunctive semantics, that is the returned answer must satisfy at least one of the return patterns.

Our second contribution consists in proposing a notion of *implementation relation,* relating a functionality needed at a peer and specified as an interface to an internal service provided by another peer. Roughly speaking, a service implements a given interface if, when called from a document for which the call pattern of the interface holds, its return (if any) satisfies one of the return patterns of the interface. Conformance of calls builds on the existing notion of query *containment*. Query containment was already studied for fragments of XPath [19, 17, 24, 21]. Conformance of returns relies on a close notion of *satisfaction* relation between patterns : pattern **P** satisfies **Q** if **Q** holds in any document that is a possible return described by pattern **P**. We show that, provided that call and return patterns of an interface are defined over disjoint sets of variables with finite domains, implementation is decidable and we study its complexity. Using this notion of implementation, we define a notion of *refinement* for a DAXML system, which consists in replacing some of its interfaces by corresponding services implementing them.

Our third contribution consists in a notion of *Distributed AXML* (DAXML) system that takes distribution over peers into account. DAXML systems are akin to GAXML systems in many respect : they possess a finite set of *peers* on which disjoint sets of internal services and disjoint parts of the AXML document are located ; However, all guards and query evaluations are performed locally to a peer, hence enforcing compartmentalization. We also introduce the notion of *external service call*, for calls to services that are owned by another peer. A external call to a service f that implements an interface consists in sending to the peer owning f a "request to execute f", that is a document containing a call to f. DAXML systems compose by doing the union of document and services, and by connecting some interfaces with a distant service implementing them via th definition of a map. With this mechanism, a peer only needs to know a distant service through the implementation map.

With this model of distribution, we show that replacing interfaces by corresponding implementations provided by new peers essentially restricts the set of possible runs of a DAXML system. Symmetrically, by replacing a service by an interface of it, we obtain a valid over-approximation of the sets of runs of a DAXML document. This gives a technique to simplify analysis of DAXML documents. Regarding analysis, we study decidability and complexity of Tree-LTL, for both global or local properties, i.e., properties that are evaluated over the successive configurations of the whole system or of the system as seen from a chosen set of peers. We show in particular that our interface theory does not add any further undecidability nor complexity to the Tree-LTL analysis of GAXML in [3]. We furthermore show that some properties of DAXML systems are preserved by composition. We also use some undecidability result for reachability to show that our notion of inmplementation is a reasonable compromise to remain decidable.

The usefulness of our DAXML model with its richer notion of interface is demonstrated on a complete example, which is a simplified model of the Dell supply chain [15]. The Dell supply chain is a very interesting example as it combines aspects of Web stored data management — the Dell Web portal — and complex distributed supplier chain involving logistics, data, and complex management. As frequently encountered in logistics applications, stocks of parts are buffered in stores

— called here the *revolvers*. The revolvers are passive peers exposing their stocks to the plant and the suppliers while not offering any service. In our DAXML modeling of the Dell supply chain, we found it very convenient to allow for shared peers and subdocument when composing DAXML systems. Direct Tree-LTL analysis of the Dell example is difficult, due to the richness of the underlying workflow and data. We illustrate the usefulness of our interface theory by showing how analysis can be simplified by using interfaces, implementations, and refinement. The Dell supply chain was already used as an example in [2], to illustrate monitoring of AXML systems, but without emphasizing the distributed nature of the example.

Several works have addressed the theme of formalisms for web services and their verification. Genest et al propose another XML-based formalism for Web services called Tree Pattern Rewriting Systems [12]. One of the main remarkable features of TPRS is that upon the assumption that all documents recorded by services are of bounded depth (i.e. they are maintained as XML trees of depth at most k, where k is given) then reachability of a given tree pattern (and consequently several other properties) is decidable. This work however does not consider the distributed nature of services, and does not deal with data.

Deutsch et al. [10] have proposed a mix of logic and model checking techniques to reason about data-intensive Web applications equipped with workflows, and have studied their composition in [11]. In this latter work, services communicate via FIFO queues. Unsurprisingly, this results in undecidability of verification in the general case. However, [11] also show decidability results under a set of restrictions allowing recursion.

This paper is organized as follows : section 2 introduces the formal background used throughout the paper. Section 3 introduces our extension of guarded AXML, and shows how this extension can be decomposed into modules and distributed. Section 4 highlights the properties of this new model. Section 5 illustrates the use of DAXML on an example, and section 6 concludes this work.

2 Background on Documents, Patterns, and Queries

In this paper, documents will be represented by trees and forests and services will be represented by queries over documents and functions transforming them. In this section we recall the corresponding basic material. Corresponding concepts are directly borrowed from the work on Guarded AXML [3], except that we do not consider continuous functions here (i.e. remanent services that can be subscribed and continuously compute values and return them to their subscribers on a push mode).

2.1 Trees, Forests, and Patterns

The material in this section reuses the usual definitions of tree, forests and patterns already introduced in [3]. Readers that are already familiar with GAXML can simply skip this part of the document. The first differences between the models appear from section 2.2. Throughout this paper, we consider unranked and unordered trees. A tree is simply a tuple $T = (N, E, \lambda_T)$ of nodes, edges and a labelling function, and a forest is a set of trees. Trees and forests are generically denoted by the symbols T and F, respectively. The notions of node, child, parent, descendant, and ancestor relations are defined in an usual way. The two relations of *child* and *descendant* are denoted by $/$ and $//$, respectively. A subtree of a tree T is the tree induced by T on the set of all descendants of a particular node.

We assume the following disjoint infinite sets : *nodes* $\mathcal{N}$ (denoted by $n, m, \dots$), *tags* Σ (denoted by $a, b, c, \dots$), *data values* $\mathcal{D}$ (denoted by $\alpha, \beta, \dots$), *data variables*

$\mathcal{V}$ (denoted by $X, Y, Z, \ldots$), and *service names* $\mathcal{F}$ (denoted by $f, g, \ldots$ or $I, J, \ldots$), possibly with subscripts. We partition the set $\mathcal{F}$ of service names as $\mathcal{F} = \mathcal{F}_{\text{int}} \uplus \mathcal{F}_{\text{ext}}$, where $\mathcal{F}_{\text{int}}$ and $\mathcal{F}_{\text{ext}}$ denote respectively *internal* and *external* service names. We also assume a special symbol denoted by $\star$, which intuitively refers to an arbitrary label. We will denote by $\mathcal{N}_T$ the nodes of a tree T. Services can be marked with the special symbols ! and ? : $\mathcal{F}^! = \{!f \mid f \in \mathcal{F}\}$ and similarly for $\mathcal{F}^?$, and we write $\bar{\mathcal{F}} = \mathcal{F} \cup \mathcal{F}^! \cup \mathcal{F}^?$.

In the sequel, we will only consider trees whose internal nodes are labeled with tags in Σ and whose leaves are labeled by either tags, labels from $\bar{\mathcal{F}}$, variables or data values. The labeling function will be generically denoted by λ, or λ_T to refer to a particular tree. We will also consider that trees are *reduced*, that is no node has isomorphic subtrees (two trees T_1 and T_2 are *isomorphic* iff there exists a bijection from the nodes of T_1 to the nodes of T_2 that preserves the child relation and the labeling of nodes). However, forests may contain several isomorphic trees. Labeled forests come up with a natural notion of *disjoint union*, denoted by $\uplus$ and we can, by abuse of notation, identify a forest with the disjoint union of its trees, seen as singleton forests : $F = \biguplus_{T \in F} T$. Such labelled trees or forests will be sometimes called (AXML) *documents*.

Definition 1 (tree pattern) *A* tree pattern *is a tuple* $P = (M, G, \lambda_M, \lambda_G)$*, where :* (M, G, λ_M) *is a tree over an alphabet* $A = \Sigma \cup \bar{\mathcal{F}} \cup \mathcal{D} \cup \mathcal{V} \cup \{\star\}$ *such that* $\lambda_M(n) \in \Sigma \cup \{\star\}$ *for every internal node* n*, and* $\lambda_G : G \to \{/, /\!/\}$*. Every variable* $X \in \mathcal{V}$ *is defined over a (not necessarily finite) domain denoted by* $Dom(X)$*.*

Tree patterns specify the shape a trees and will be used to query a document. Let P be a tree pattern. A node n of P such that $\lambda_M(n) = a$ designates a node tagged by a. When $\lambda_M(n) = \star$, then the node can take any value or tag in $\Sigma \cup \bar{\mathcal{F}} \cup \mathcal{D} \cup \mathcal{V}$. For an edge (x, y) of P, if $\lambda_G(x, y) = /$, then y must be a child of x. If $\lambda_G(x, y) = /\!/$, then y must be a descendant of x.

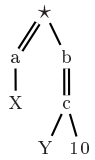


FIG. 1 – An example of tree pattern

Figure 1 shows an example of tree pattern. This pattern describes trees that contain a node tagged by a. This node must have a child that is a leaf, and the value attached to this leaf is associated to variable X. In addition to this first constraint, the root of every tree satisfying this pattern has a child tagged by b. This child must have a descendant tagged by c which has a child which is a leave with value 10, and a child which value is associated to variable Y. Note that nothing forces these latter leaves to be distinct nodes. This definition differs slightly from the usual definition of tree patterns (see [24, 21, 17] for example), in the sense that it allows some nodes to be labelled by variables. It has been shown [17] that tree patterns are equivalent to a fragment of XPath allowing the use of labels $\sigma \in \Sigma$, child relation, descendant relation, filtering, and $\star$. Note that the full XPath contains more concepts such as nodeset equality or even arithmetic operators, that will not be considered in this paper. Tree patterns are used to describe the general shape of a tree, and the property whether a tree conforms to this shape is captured by the notion of matching.

Definition 2 (matching) *Let $P = (M_P, G_P, \lambda_M, \lambda_G)$ be a tree pattern and $T = (N, /, \lambda_T)$ a tree. A* matching *of P into T is a mapping μ from the nodes of P to the nodes of T such that :*

1. *the root of P is mapped to the root of T,*
2. *the descendant and child relations of the tree pattern are respected by μ, that is for every edge (n,m) of P, if $\lambda_G(n,m) = /$, then $\mu(m)$ is a child of $\mu(m)$ in T, and if $\lambda_G(n,m) = //$, then $\mu(m)$ is a descendant of $\mu(n)$ in T.*
3. *μ preserves the labels, that is for every node n of P such that $\lambda_M(n) \in \Sigma \cup \bar{\mathcal{F}} \cup \mathcal{D}$, $\lambda_M(\mu(n)) = \lambda_M(n)$. Note that this does not apply to nodes of P labelled by $\star$, which can be mapped by μ to nodes labelled by any symbol or value ;*
4. *Nodes labeled with variables are mapped to data values. In particular, this enforces nodes tagged by variable names to be mapped to leaves of T. The image of variable nodes must belong to the domain of variables, that is if $\lambda_M(n) = X \in \mathcal{V}$ then $\lambda_T(\mu(x)) \in Dom(X)$. Furthermore, if two nodes are labelled by the same variables, then μ sends them onto nodes of T with the same values. We will frequently denote by $\mu(X)$ the unique symbol or data value associated by μ to nodes labelled by variable X.*

We will say that a tree pattern P holds *in a tree T, written $T \models P$, when there is at least one matching from P to T.*

Let $\mathcal{X}$ be a set of variables. A *valuation* of $\mathcal{X}$ is a function $v : \mathcal{X} \mapsto \mathcal{D}$ that associates a value to each variable in $\mathcal{X}$. For a given boolean combination *cond* of expressions on variables of $\mathcal{X}$, we will say that v is *consistent* with *cond* if and only if predicate *cond* holds for values provided by v. For a given tree pattern P, we will denote by $\mathcal{X}_P$ the set of its variables, and by P_v the pattern obtained by replacing each node labeled by a variable $X \in \mathcal{X}_P$ by its valuation $v(X)$. A mapping μ from a tree pattern P to a tree T defines a valuation $v_\mu : \mathcal{X}_P \mapsto \mathcal{D}$ that associates to each variable $X \in \mathcal{X}_P$ located on a node n the value $\lambda(\mu(n))$. Mapping μ is *consistent* with a boolean condition *cond* if and only if v_μ is consistent with *cond*. These definitions extend to sets of tree patterns.

Definition 3 (pattern and pattern matching) *A* pattern *is a pair*

$$\mathbf{P} = (\mathbb{P}, cond) \tag{1}$$

where $\mathbb{P}$ is a finite set of tree patterns and cond is a Boolean combination of expressions of the form $X \sim \alpha$, where $X \in \mathcal{V}$, $\alpha \in \mathcal{V} \cup \mathcal{D}$, and $\sim \in \{=, \neq, \leq\}$. We thus assume that the domain of all variables that appear in an expression of the form $X \leq \alpha$ or $Y \leq X$ is ordered. As for forests, sets of tree patterns come up with a disjoint union, denoted by $\uplus$ and we can write, by abuse of notation, $\mathbb{P} = \biguplus_{P \in \mathbb{P}} P$.

Let $\mathbf{P} = (\mathbb{P}, cond)$ be a pattern, and let F be a forest. A matching *of $\mathbf{P}$ into F is a mapping μ that is a matching of each $P \in \mathbb{P}$ into some tree of F, and for which cond is satisfied. Formally,*

$$\mu : \biguplus_{P \in \mathbb{P}} \mathcal{N}_P \;\mapsto\; \biguplus_{T \in F} \mathcal{N}_T$$

satisfies cond, and for each $P \in \mathbb{P}$, the restriction of μ to any $P \in \mathbb{P}$ is a matching of P into some $T \in F$. Say that a pattern $\mathbf{P}$ holds *in a forest F, written $F \models \mathbf{P}$, iff either $F = \emptyset$ or there exists at least one matching of $\mathbf{P}$ into F.*

This definition of matching relies on the usual notion of tree pattern homomorphism, but also considers that empty forests satisfy **any pattern**. The reason for this is that patterns will be compared to the values (forests) returned by a function, and that we will need to consider functions returning empty forests. Note that nothing

forces a matching to be an injective mapping. Hence, the same node of a tree (and as a consequence the same paths) can be used as an image of different nodes in the pattern. Similarly, when considering matchings on forests, two patterns can be mapped to nodes of the same tree. Consider for example the pattern of Figure 1. A node with value 10 in a tree can be the image of the two children of node with tag *c*. Consider a pattern $\mathbf{P} = (P, cond)$ where P is the tree pattern of Figure 1, and $cond = \{X \leq Y\}$. Figure 2 illustrates a matching that maps nodes of P onto nodes

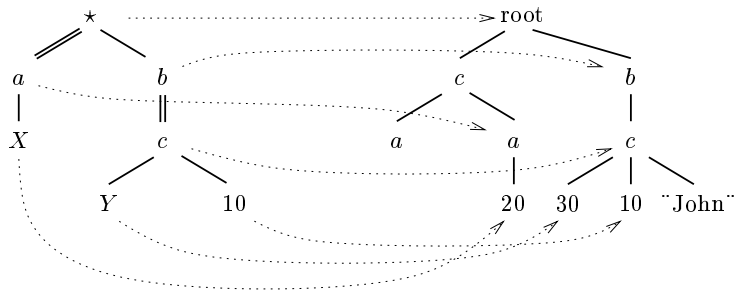


FIG. 2 – An example of matching

of a tree and satisfies *cond*. The matching relation μ is figured by dotted arrows. Note that on this example, due to the constraint, Y cannot be mapped to a node with value 10.

The definition of matching extends to boolean combinations of patterns as follows. Write

$$\begin{array}{lll} F \models \mathbf{P} \wedge \mathbf{P}' & \text{if and only if} & F \models \mathbf{P} \wedge F \models \mathbf{P}' \\ F \models \mathbf{P} \vee \mathbf{P}' & \text{if and only if} & F \models \mathbf{P} \vee F \models \mathbf{P}' \\ F \models \neg\mathbf{P} & \text{if and only if} & \text{there exists no matching from } \mathbf{P} \text{ to } F \end{array} \tag{2}$$

In particular, we have that *any negated pattern* $\neg\mathbf{P}$ holds on the empty forest $F = \emptyset$. Hence, the empty forest satisfies any boolean combination of patterns. This choice will be justified later by the fact that functions are allowed to return empty forests. Note that for boolean combinations of patterns $\mathbf{P}_1, \ldots, \mathbf{P}_k$, $F \models \mathbf{P}_i$ is evaluated separately for each $i \in 1..k$. In particular, this means that when a variable X appears both in $\mathbf{P}$ and $\mathbf{P}'$ in a boolean combination, it should be considered as two different variables $X_{\mathbf{P}}$ and $X_{\mathbf{P}'}$. We can nevertheless enforce equality of variables with an additional constraint of the form $X = Y$. Note also that conjunction of tree patterns can be seen as syntactic sugar, as $P_1 \wedge P_2$ holds in a forest F if and only if the pattern $\mathbf{P} = (\mathbb{P}_1 \uplus \mathbb{P}_2, cond_1 \wedge cond_2)$ holds for F.

In some cases we will use patterns that are evaluated relatively to a specified node in a tree. More precisely, a *relative pattern* is a pair

$$(\mathbf{P}, self) \tag{3}$$

where $\mathbf{P}$ is a pattern and *self* is a node of $\mathbf{P}$. A relative pattern $(\mathbf{P}, self)$ is evaluated on a pair (F, n) where F is a forest and n is a node of F. The evaluation of a pattern $(\mathbf{P}, self)$ on (F, n) for candidate matchings forces the node *self* in the pattern to be mapped to n. If such matchings exist, we say that $(\mathbf{P}, self)$ *holds* in (F, n), written

$$(F, n) \models (\mathbf{P}, self).$$

In contrast to the matching of patterns into forests, the case $F = \emptyset$ is not considered as it makes no sense as soon as a node n of F is selected.

Notation : For convenience, we shall sometimes simply write $\mathbf{P}$ instead of $(\mathbf{P}, self)$ for a relative pattern, if no confusion can result.

Theorem 1 *Let T be a tree, and $\mathbf{P} = (\mathbb{P}, cond)$ be a tree pattern. Testing if $T \models \mathbf{P}$ is NP-complete.*

Proof sketch: The complete proof is provided in Appendix A. One can chose a mapping from variables to leaves and verify the condition in polynomial time. Then, checking the rest of the pattern can be done with classical algorithms for patterns without variables ([13]). The hardness part can be shown by a reduction from 3SAT. □

This NP-completeness result extends to forests, and to conjunction or disjunction of patterns. Of course, checking $\neg\mathbf{P}$ will be co-NP complete. So far, we made no hypotheses on the shape of patterns or on their variables. However, we can consider several hypotheses to reduce complexity of pattern satisfaction or to allow deciding the compatibility of a service and an interface.

1. *child-only* hypothesis : $\mathbf{P}$ satisfies this criterion if it does not use descendant edges.
2. *Finite-domain* hypothesis : $\mathbf{P}$ satisfies this criterion if all its variables range over a finite domain. We will show in the sequel that the finite domain hypothesis is needed to define an effective notion of interface and implementation.

For a given variable X with finite domain, we will denote by $\|X\|$ the number of bits needed to encode a valuation of X. Note that the number of consistent valuations for a set of variables and constraints can be exponential in the number of variables. Fortunately, we do not need to consider all possible valuations if the objective is to check whether $T \models P$. Define the *size of variable sets* of tree patterns and patterns as :

$$\|P\| = \sum_{X \in \mathcal{X}_P} \|X\| \text{ for a tree pattern } P, \text{ and} \tag{4}$$

$$\|\mathbf{P}\| = \sum_{P \in \mathbb{P}} \|P\| \text{ for a pattern } \mathbf{P} = (\mathbb{P}, cond). \tag{5}$$

2.2 Queries

Patterns are used to ensure that a forest has a certain shape, but cannot be used to produce new documents as the result returned by the evaluation of a pattern is only *true* (there exists a mapping from $\mathbf{P}$ to F) or *false* (there is no consistent mapping from $\mathbf{P}$ to F). To propose more powerful services, we need a query mechanism to build structured answers and produce a new document out of an existing one. Queries are non-deterministic mechanisms returning a forest from an input document. They will constitute the main mechanism used to implement services in our framework.

Definition 4 (queries) *A (non-deterministic)* query *is an expression*

$$Q = Body \rightarrow Head, \tag{6}$$

where $Body = (\mathbb{P}_B, cond_B)$ and $Head = (\mathbb{P}_H, cond_H)$ are patterns defined respectively over sets of variables $\mathcal{X}_B$ and $\mathcal{X}_H$, and such that for each tree of Head :

- *internal nodes have labels in Σ and leaves have labels in $\Sigma \cup \mathcal{F}^! \cup \mathcal{V} \cup \mathcal{D}$;*
- *all edges are labeled "child",*
- *there is one designated node c, called the* constructor *node, such that the subtree rooted at c contains all variables of this tree.*

The variables of Head that do not occur in Body, are called free variables*, the other variables are designated as* bound variables*. When Head does not contain free variables, Q is called a* deterministic *query. When Head only involves a trivial condition, Q is said to be* unconstrained*.*

We furthermore require consistency of constraints $cond_B$ and $cond_H$: for every consistent valuation of variables in $\mathcal{X}_B$ there is a possible consistent valuation of variables in $\mathcal{X}_H$:

$$\forall v : v \models cond_B \implies \exists v' : \forall x \in \mathcal{X}_B \cap \mathcal{X}_H, v(x) = v'(x) \text{ and } v' \models cond_H \qquad (7)$$

As for patterns, we can consider queries evaluated relatively to a specified node in the input tree. A relative query *is defined like a query, except that its body is a relative pattern.*

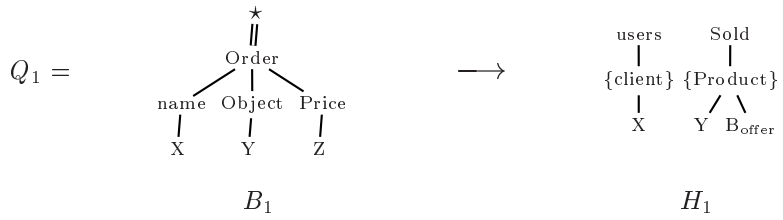


FIG. 3 – An example of nondeterministic query

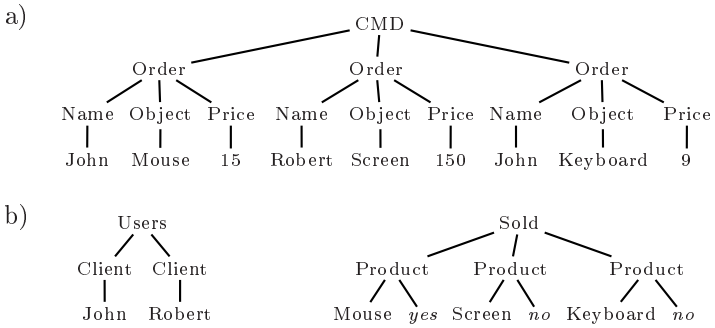


FIG. 4 – a) An example document T, and b) a result of query $Q_1(T)$

Constructor nodes will be used during query evaluation to group different answers to a query in a single subtree. We will frequently represent queries as in Figure 3. The body and head parts (in the example B_1 and H_1) of the query are represented as trees, and separated by an arrow. The constructor nodes in the Head part are distinguished by brackets (on the example of figure 3, there are two constructors, labeled by $\{client\}$ and $\{Product\}$). The range of variables X and Y is the set of strings of at most 15 characters, and the domain of free variable B_{offer} is $\{yes, no\}$. The query depicted in figure 3 can be used to recover client names and ordered products from a store. When applied to a document containing orders, such as the tree of figure 4-a), this query returns the set of all clients, and the set of all sold products (represented as trees), as shown in figure 4-b). The free variable B_{offer} states whether or not an offer is made on the considered product.

Querying a document :

Let us now explain how queries operate on documents. Intuitively, the *Body* is used to select the values of bound variables from an input document, and the *Head* defines the structure of the answer, where bound variable values are those selected by the *Body*, and free variables are replaced by nondeterministically chosen values.

Formally, let F be a forest and $Q = Body \rightarrow Head$. Let $\mathcal{M}$ be the (finite) set of matchings of *Body* into F.

1. For each tree H of *Head*, let c be the constructor node of H and H_c be the subtree of H rooted at c. For each matching $\mu \in \mathcal{M}$, let $\mu(H_c)$ be an isomorphic copy of H_c with new nodes, in which every *bound* variable X occurring in H is replaced by $\mu(X)$.
2. The forest $\{\mu(H_c) \mid H \in Head\}$ still contains free variables, namely in $\mathcal{X}_H \setminus \mathcal{X}_B$; a valuation $v \in \Pi_{X \in \mathcal{X}_H \setminus \mathcal{X}_B} Dom(X)$ for them is selected subject to condition $cond_H$ of pattern *Head* (that is such that $\mu \cup v \models cond_H$), free variables are replaced by their valuation, and the trees of the resulting forest are all reduced. Call $\{\bar{\mu}(H_c) \mid H \in Head\}$ the resulting forest consisting of reduced, fully valued, trees.
3. For each tree H of *Head*, replace the constructor node c (and hence the whole subtree rooted at c) by $\{\bar{\mu}(H_c) \mid \mu \in \mathcal{M}\}$ and call $Q_H(F)$ the resulting forest. The result $Q(F)$ of applying query Q to F is then

$$Q(F) \;=\; \biguplus_{H \in Head} Q_H(F)$$

For Q a relative query, the *Body* is evaluated for its possible matchings on a pair (F, n) where F is a forest and n is a node of F. Corresponding (nondeterministic) returns are denoted by

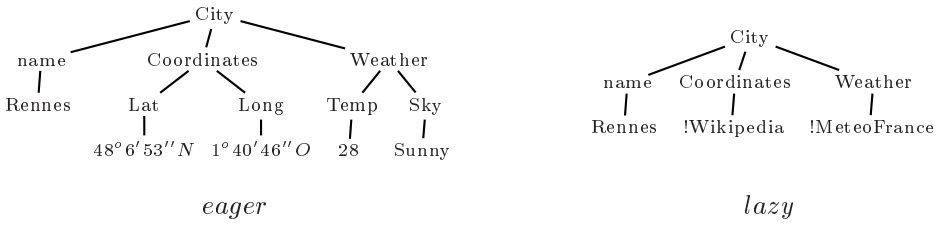
$$Q(F, n)$$

3 The Distributed Guarded AXML Model

As suggested by its name, Active XML [1] consists in XML documents augmented by actions. The philosophy behind AXML is that services or queries on documents may have an eager interpretation, and return trees with the required data, or have a lazy intentional interpretation, and return trees containing references to services that will help completing the answer to the asked question. Consider for instances a service *CityDescription* that returns structured information (geographical location, weather,...) for a given city passed as a parameter of a call. An eager interpretation of such service is a mechanism that returns a structured document with data on leaves, as for instance in the leftmost tree of Figure 5. Another possibility is to return a tree with the address of a service providing the geographical information, and a pointer to a weather forecast service, as in the rightmost tree of Figure 5.

Guarded Active XML [3] control flows in AXML via guarded services. DAXML aims at modeling distributed systems, where agents (called peers) manage data, but also provide and require services to one another. In a service oriented architecture, when a peer must use a service, it may either own this service or borrow it from another peer. For this reason, we will distinguish internal and external services for each peer. The overall description of a DAXML system is formalized through the concept of DAXML Schema, which lists peers, external and internal services, and the relations among them.

In guarded AXML [3], internal services are captured by the concept of "internal function", whereas external services are captured by the notion of "external function", and specified by means of DTDs that depict the shape of returned forests. In



eager *lazy*

FIG. 5 – Eager and lazy interpretation of a service

DAXML internal services are also represented by functions, but the notion of external function is defined using an *interface* rather than DTDs. An interface specifies the allowed parameters and the possible returns of a call. As usually in service oriented architectures, it must be possible to relate an interface depicting an external service to its *implementation* by the peer offering that service. We propose such a notion of implementation based on the known concept of query containment [24].

The distributed nature of DAXML is highlighted by the possibility of associating, to an external service specified as an interface, an implementation for it offered by another peer. A peer calling an external service can compose with the peer offering that service by associating, to the service interface, its offered implementation. This natural mechanism is formalized as a new notion of *composition* of DAXML schemas.

One advantage of this approach is the flexibility it offers regarding analysis. When performing analyses of how a (distributed) schema can transform a given (distributed) document — also called an *instance* —, services can be represented at will by their implementations, or by more abstract interfaces. This allows exchanging simplicity for power in a tunable way when performing such analyses.

3.1 Services as Functions and Interfaces

In this section we introduce basic notions to represent services, namely via *functions* and *interfaces*. Following [3], the concept of "internal service" is formalized through the notion of *function*. Roughly speaking, a function is a mechanism that takes a tree as input, performs some computation, and returns a forest of AXML documents. Our functions slightly extend those of [3] by allowing some nondeterminism in the shape of returned values, and in the valuations of some free variables that appear in patterns. Note that using free variables in patterns was already proposed in [3], but that the role of these variables was to allow for existential quantifiers $\exists X.P$ in logical formulæ over patterns.

3.1.1 Functions

In the rest of the paper, we will distinguish internal and external function names using elements of the $\mathcal{F}_{\text{int}}$ and $\mathcal{F}_{\text{ext}}$ alphabets. We will furthermore suppose that $\mathcal{F}_{\text{int}}$ and $\mathcal{F}_{\text{ext}}$ are disjoint.

Definition 5 (function) *A* function *is a tuple*

$$\left(f\,,\,G_f^c\,,\,Q_f^c\,,\,\left\{(G_f^r[k],Q_f^r[k])\right\}_{k\in K_f}\right)$$

where K_f is a finite set of indexes, $f \in \mathcal{F}_{\text{int}}$ is a function name and :
- *G_f^c is a boolean combination of (relative) patterns, called the* call guard *;*
- *Q_f^c is a relative query, called the* call query *;*

- $\{(G^r_f[k], Q^r_f[k])\}_{k\in K_f}$ *is a finite set of* guarded return queries *consisting, for each* $k \in K_f$*, of :*
  - *a boolean combination of patterns* $G^r_f[k]$*, called the* return guard*, and*
  - *a query* $Q^r_f[k]$*, called the* return query*.*

Note that we do not require the guards $G^r_f[k], k \in K_f$ *to be pairwise disjoint. Whenever convenient, a function will simply be referred to by its name, seen as a unique identifier.*

The role of DAXML functions is to query and transform AXML documents or more precisely document instances, that contains structured data plus information on ongoing service evaluations. If the call and return queries are both deterministic and unconstrained, and if a function has a single return query, the above definition corresponds to that of internal function in [3]. The differences with the function model proposed in [3] are minor : we allow multiple returns (intuitively, K indexes a finite set of possible returns of the function), and non deterministic values in the queries heads. However, all these new features can be emulated in the original AXML model by additional services that introduce non determinism.

In a DAXML system, each peer p in the system manages structured data, under the form of AXML forests. Trees in this forest contain leaves with labels of the form $!f$, where f is either a internal or external function name. Intuitively, when a tree contains a node tagged y a service name, it means that the data contained in the tree can be completed by the values returned after a call to service f. Internal functions are used locally by the peer owning them, and applied locally on the document owned by this peer. Functions evaluations are performed into three phases : a call to the function, the internal computation steps, and the return. A call to a function by a peer will create a new temporary tree called the *workspace*. This workspace will then be transformed by successive calls to other services. When a guard $G^r_f[k]$ becomes true, then the query $Q^r_f[k]$ is applied to the workspace. Hence, at a given moment, a peer may own its structured data, plus a set of temporary workspaces created for functions evaluations. Of course, each workspace is related to an instance of a service call. To avoid multiple evaluation of the same instance of a service call, nods at which a service is called will be tagged differently : $!f$ if at this node service f can be called to obtain some information, $?f$ if service f has been called, and the system is waiting for the value returned by f (hence there exists a workspace in which this instance of the service is being evaluated), and f if the service call is complete (a value has been returned, and the temporary workspace attached to this call removed). This setting is captured by the notion of *document instance*.

Definition 6 (document instance) *A* document instance *over s et of peers* $\mathcal{P}$ *is a tuple* $D = (F, eval)$*, where* F *is a forest and* eval *is an injective function over the subset of nodes in* F *that are labeled with* $?f$ *for some* $f \in \mathcal{F}_{\text{int}}$ *and such that :*

1. *For each node* n *with label* $?f$*,* $eval(n)$ *is a tree in* F *with root label* a_f*, called a* workspace*;*
2. *Every tree in* F *with root labeled* a_f *is* $eval(n)$ *for some* n *labeled by* $?f$*.*

Intuitively, document instances can be seen as the local state of a peer. We are now ready to define formally how internal functions transform document instances. We will say that there is an *internal move* from $D = (F, eval)$ to $D' = (F', eval')$ through $f \in \mathcal{F}_{\text{int}}$, written

$$D \vdash^f_{\text{func}} D' \tag{8}$$

if one of the following cases hold :

– *Call* : there exists some node n in $T \in F$, labeled by $!f$, such that $(F, n) \models G_f^c$; F' is obtained by changing the label of n to $?f$ and adding to the graph of *eval* the pair (n, T'), where T' is a tree consisting of a root a_f connected to the forest $Q_f^c(F, n)$; *eval*$'$ extends *eval* with the pair (n, T'). This is illustrated in figure 6.

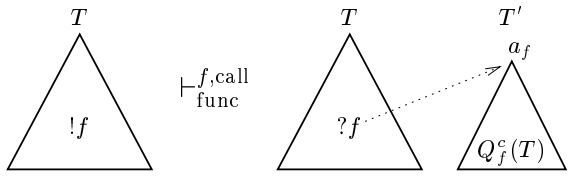


FIG. 6 – Internal move : function call

– *Return* : there exists some node n in F, labeled by $?f$, and some $k \in K_f$, such that $eval(n) \models G_f^r[k]$ and contains no node labeled with $?g$ for some $g \in \Phi$ (contains no running call). Then, D' is obtained from D as follows :

1. evaluate $Q_f^r[k](eval(n))$ and add the result as a sibling of node n ;
2. remove $eval(n)$ from F and remove n from the domain of *eval* ;
3. change the label of n to f (with no mark)

Observe that, since guards may not be pairwise disjoint, the selection of $k \in K_f$ may be nondeterministic. This is illustrated in figure 7. Another important observation is that the returned forest may contain trees labelled by any symbols in $\Sigma \cup \mathcal{F}^! \cup \mathcal{D}$. This allows returning references to another service (lazy evaluation policy) instead of recursively calling services before returning a completely evaluated tree (eager evaluation policy).

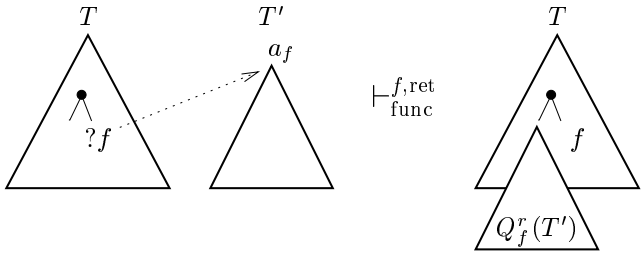


FIG. 7 – Internal move : function return

Whenever needed we shall distinguish a call from a return by writing

$$D \vdash_{\text{func}}^{f,\text{call}} D' \quad \text{and} \quad D \vdash_{\text{func}}^{f,\text{ret}} D',$$

respectively. We simply write $D \vdash_{\text{func}} D'$ to mean $D \vdash_{\text{func}}^{f} D'$ for some $f \in \mathcal{F}_{\text{int}}$.

3.1.2 Interfaces

Peers in DAXML systems own internal and external services. We have detailed in section 3.1.1 how internal functions modify the local states of peers owning them. The evaluation of internal services relies mainly on queries. The evaluation mechanisms is not explicitly described for external services, which are perceived by a peer as an interface. An interface specifies the allowed parameters and the possible returns of a call. It can be defined formally as follows :

Definition 7 (interface) *An AXML* interface *is a tuple*

$$\left(I, \mathbf{P}_I^{\mathrm{C}}, \{\mathbf{P}_I^{\mathrm{R}}[k]\}_{k\in K_I}\right)$$

where $I \in \mathcal{F}_{\mathrm{ext}}$ is its name, $\mathbf{P}_I^{\mathrm{C}}$ is a relative pattern defined over set of variables $\mathcal{X}_I^{\mathrm{C}}$, and $\{\mathbf{P}_I^{\mathrm{R}}[k]\}_{k\in K_I}$ is a set of patterns defined over set of variables $\mathcal{X}_I^{\mathrm{R}}$ such that $\mathcal{X}_I^{\mathrm{C}} \cap \mathcal{X}_I^{\mathrm{R}} = \emptyset$. We require in addition that, for each tree of each $\mathbf{P}_I^{\mathrm{R}}[k]$:

- *internal nodes have labels in Σ and leaves have labels in $\Sigma \cup \mathcal{F}^! \cup \mathcal{V} \cup \mathcal{D}$;*
- *there is one designated node c, called the* constructor *node, such that the subtree rooted at c contains all variables of this tree.*

Whenever convenient, an interface will simply be referred through its name, seen as a unique identifier. We will frequently call $\mathbf{P}_I^{\mathrm{C}}$ the call pattern *and $\{\mathbf{P}_I^{\mathrm{R}}[k]\}_{k\in K_I}$ the* return patterns *of interface I,*

An interface specifies at a calling peer the input that must be accepted by a distant service, and the set of "all possible returns" received after an external call. We formalize this next. As services are non-deterministic, the interface may also specify a finite set of return patterns, $\{\mathbf{P}_I^{\mathrm{R}}[k]\}_{k\in K_I}$. Let $\mathbf{P}_I^{\mathrm{R}} = (\mathbb{P}, cond_P)$ be one of the return patterns of an interface I. The set $[\mathbf{P}_I^{\mathrm{R}}]$ of the *possible returns of* $\mathbf{P}_I^{\mathrm{R}}$ is defined as follows, where $\{\!\{cond_P\}\!\}$ is the set of all valuations $v \in \Pi_{X\in\mathcal{X}_I^{\mathrm{R}}} Dom(X)$ of the variables of $\mathbf{P}_I^{\mathrm{R}}$ that are consistent with $cond_P$. :

1. For each tree pattern P of $\mathbb{P}$, let c be the constructor node of P and P_c be the subtree of P rooted at c. For each $v \in \{\!\{cond_P\}\!\}$, let $v(P_c)$ be an isomorphic copy of P_c with new nodes, in which every variable X occurring in P is replaced by $v(X)$, and that is then reduced. A pattern $v(P_c)$ will be called a *pattern valuation.*
2. For each tree pattern P of $\mathbb{P}$ and each subset $V \subseteq \{\!\{cond_P\}\!\}$, replace the whole subtree rooted at c by the forest $\{v(P_c) \mid v \in V\}$. Call $V(P)$ the result and define the following pattern $V(\mathbf{P}_I^{\mathrm{R}}) = \{V(P) \mid P \in \mathbb{P}\}$, which possesses neither condition nor variables.
3. Finally, the *set of possible return patterns* of $\mathbf{P}_I^{\mathrm{R}}$ is the set of patterns defined as follows :

$$[\mathbf{P}_I^{\mathrm{R}}] = \left\{ V(\mathbf{P}_I^{\mathrm{R}}) \,\middle|\, V \subseteq \{\!\{cond_P\}\!\} \wedge |V| \neq \emptyset \right\} \tag{9}$$

With this definition, the set of possible returns depicted by a pattern need not be finite, as it can be a forest of arbitrary size with trees of arbitrary width. However, under the finite domain hypothesis, the size of $[\mathbf{P}_I^{\mathrm{R}}]$ remains finite (note however that it can be doubly exponential in $\|\mathbf{P}_I^{\mathrm{R}}\|$ - as defined in (5)-).

The main idea behind interfaces is that these external descriptions of inputs and outputs will be implemented by internal functions. Hence, we have to consider the possibility that a return query of a function returns an empty set of answers. For this reason we also need to consider a specific pattern that describes the structure of the pattern above constructor nodes. For a tree pattern P with constructor node, we will denote by $P_\emptyset = V_\emptyset(P)$ the pattern obtained by removing the constructor node and its subtree from P. This definition easily extends to patterns by removing constructors and subtrees in all tree patterns composing the pattern). Observe that if P is the head of a return query $Body \rightarrow P$, $P_\emptyset$ corresponds to the returned value when no matching exists for $Body$.

Let us consider the pattern P of figure 8 with condition $cond = \{X = true \wedge Y \in \{1, 2\}\}$. This pattern contains a constructor node, denoted by $\{Record\}$. According

to *cond*, there are two possible valuations for X, Y. The possible returns $[P]$ computed from P with respect to *cond* are represented in Figure 9. They are built by replacing the subtree of P rooted at $\{Record\}$ by a subset of similar trees where X and Y are respectively replaced by *true* or a value in $\{1, 2\}$. In this example $P_\emptyset$ consists in a single node tagged by $PersonalData$

P =

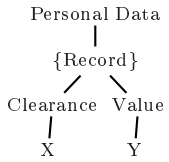


FIG. 8 – An example of pattern with constructor

$[P]$ =

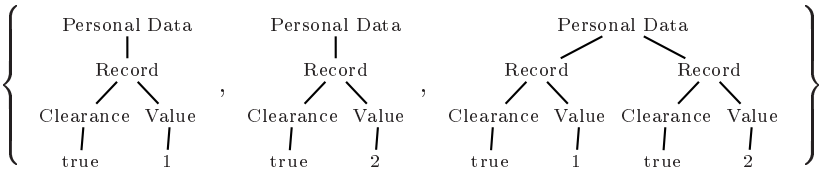


FIG. 9 – Possible return patterns $[P]$ for $cond = \{X = true \wedge Y \in \{1, 2\}\}$

The set of possible returns is used to depict the set of legal values that a service implementing an interface may return. Hence, a forest F returned by a services will need to satisfy one of the patterns listed in the set of possible return patterns.

Definition 8 *Let* $\left(I, \mathbf{P}_I^{\mathrm{C}}, \{\mathbf{P}_I^{\mathrm{R}}[k]\}_{k \in K_I}\right)$ *be an interface, and let* F *be a forest. We will say that* F satisfies *return pattern* $\mathbf{P}_I^{\mathrm{R}}[k]$ *and write*

$$F \models \left[\mathbf{P}_I^{\mathrm{R}}[k]\right] \tag{10}$$

if and only if there exists a pattern $\mathbf{P} \in \left[\mathbf{P}_I^{\mathrm{R}}[k]\right]$ *such that* $F \models \mathbf{P}$*. We will say that* F *is a* legal return *for interface* I *if and only if there exists* $k \in K_I$ *such that* $F \models \left[\mathbf{P}_I^{\mathrm{R}}[k]\right]$

Note that F only needs to satisfy one of the patterns listed in $\left[\mathbf{P}_I^{\mathrm{R}}\right]$. Hence, $\left[\mathbf{P}_I^{\mathrm{R}}\right]$ can not be seen as a pattern, but rather as a **disjunction** of patterns. Note also that as we require that $F \models \mathbf{P}$ for some $\mathbf{P} \in \left[\mathbf{P}_I^{\mathrm{R}}\right]$, the forests F returned by a service do not need to be isomorphic to one of the patterns in $[P]$ to be legal return values, and can then contain **more information** than depicted in the satisfied pattern $\mathbf{P}$.

Note also that we do not have to compare a forest with the patterns obtained from all combinations of valuations. Let $[\mathbf{P}]$ be the possible returns computed from a pattern $\mathbf{P} = (\mathbb{P}, cond_P)$. We can define subsets of $[\mathbf{P}]$ according to the size of the valuation sets chosen in $\{\!\{cond_P\}\!\}$. Let us write $\left[\mathbf{P}^i\right] = \{\, V(\mathbf{P}) \mid V \subseteq \{\!\{cond_P\}\!\} \wedge |V| = i \,\}$.

Theorem 2 *Let* $\mathbf{P} = (\mathbb{P}, cond_P)$ *be a patterns with constructor nodes. Then* $F \models [\mathbf{P}]$ *if and only if* $F \models \left[\mathbf{P}^1\right]$

In section 3.1.1, we have discussed how internal functions operate on document instances. External function are defined as interfaces, and express the needs of a peer, that are fulfilled by its environment. We next define the action of interfaces

on documents instances. External function will operate like functions on document instances, with the exception that, as the computation for an external service is performed by the environment of the calling peer, no workspace is created. The evaluation of an external service can then be decomposed into two phases : the call and the return.

Say that there is an *external move* from $D = (F, eval)$ to $D' = (F', eval')$ through interface $\left(I, \mathbf{P}_I^{\mathrm{C}}, \{\mathbf{P}_I^{\mathrm{R}}[k]\}_{k \in K_I}\right) \in \mathcal{F}_{\mathrm{ext}}$, written

$$D \vdash_{\mathrm{intf}}^{I} D' \tag{11}$$

if $eval' = eval$ and one of the following cases hold :

- *Call* : there exists some tree $T \in F$ located of p and containing a node n labeled by $!I$, such that $(T_n, n) \models \mathbf{P}_I^{\mathrm{C}}$, where T_n is the smallest subtree of T containing n and its siblings; Here, T_n represents de parameters of the external call. The new forest F' is obtained by changing, in F, the label of n to $?I$. Here, $\mathbf{P}_I^{\mathrm{C}}$ plays the role of a guard evaluated on the calling peer, that guarantees that the parameters of a call to an external function can be found on the part of the document owned by the calling peer.
- *Return* : there exists some node n in F, labeled by $?I$; F' is obtained by changing, in F, the label of n to I (with no mark) and by adding as a sibling of n some forest satisfying a pattern in $\left[\mathbf{P}_I^{\mathrm{R}}[k]\right] \cup (\mathbf{P}_I^{\mathrm{R}}[k])_{\emptyset}$ for some k non deterministically chosen in K_I.

Note that without the finite domain and child only assumption, there is an infinite number of forests that can be appended to the document instance. However, with the finite domain assumption, one can easily bound this set assuming some knowledge on the depth of the trees returned by a service that implements I (functions always return bounded depth forests). Whenever needed we shall distinguish a call from a return by writing

$$D \vdash_{\mathrm{intf}}^{I,\mathrm{call}} D' \quad \text{and} \quad D \vdash_{\mathrm{intf}}^{I,\mathrm{ret}} D',$$

respectively. We simply write $D \vdash_{\mathrm{intf}} D'$ to mean $D \vdash_{\mathrm{intf}}^{I} D'$ for some $I \in \mathcal{F}_{\mathrm{ext}}$.

3.2 Implementation

A site will use a distant service if and only if the provided implementation fits the interface description. For AXML, this informal notion of "fitting an interface" is formalized using the two concepts of *containment* (for the call) and *satisfaction* (for the return). We begin with containment.

3.2.1 Containment

Containment is a way to compare and optimize queries. This problem is considered almost systematically when a new query language is proposed. An extensive literature exists on containment. We refer interested readers to [24] for a survey on tree patterns containment.

Definition 9 (containment) *A (relative) pattern* $\mathbf{P}'$ *is* contained *in a (relative) pattern* $\mathbf{P}$*, written* $\mathbf{P}' \subseteq \mathbf{P}$*, iff, for every tree* T*,* $T \models \mathbf{P}'$ *implies* $T \models \mathbf{P}$*.*

The pattern containment proposed above is exactly the usual notion of boolean query containment, that was widely studied in the literature (see for instance [24] for a survey of this domain). Several notions of containment exist, but as most of containment problems can be brought back to a boolean containment, the latter is the

most studied notion. The complexity of containment changes depending on the fragment of the query language that is used. For XPath, it has been shown [24, 21, 17] that, depending on the fragment, containment can be a polynomial, Co-NP or even an Exptime problem. When one requires that containment is restricted to queries satisfying some integrity constraints (SXICs) and a DTD, containment can even become undecidable. Undecidability results also hold under simple DTDs when the considered fragment of XPath allows testing nodesets [20]. Containment is a crucial notion for our definition of interfaces and implementations. Implementation must remain an effective problem, which justifies some restrictions on the Service architecture proposed in this paper. Here, we consider containment of patterns which corresponds to containment for a fragment of XPath that uses only child and descendant relation, filtering and wildcards. The difficult and expensive point will be to deal with variables and constraints.

Theorem 3 *Let* $\mathbf{P}$ *and* $\mathbf{P}'$ *be two patterns whose variables range over infinite domains. Then containment of* $\mathbf{P}$ *in* $\mathbf{P}'$ *is an undecidable problem.*

Proof sketch: The proof is an adaptation of the proof in [5] for the undecidability of containment for conjunctive queries with variables equality and inequality. For each instance of the PCP, one can compute two patterns $\mathbf{P}$ and $\mathbf{P}'$ such that $\mathbf{P} \subseteq \mathbf{P}'$ iff there is no solution to the considered instance of the PCP. P is used to encode the shape of potential PCP solutions. P' is used to collect a list of bad properties of trees that can not be PCP solutions. The details of the encoding are provided in the appendix. □

This undecidability of containment for patterns with variables over infinite domains holds as soon as equality and inequality of data values are allowed in patterns. However, with the finite domain assumption, the undecidability result does not hold anymore.

Theorem 4 *Let* $\mathbf{P}$ *and* $\mathbf{P}'$ *be two patterns whose variables range over finite domains. Then, deciding whether* $\mathbf{P} \subseteq \mathbf{P}'$ *is in* Co-NexpTime.

Proof sketch: To provide a counter example T such that $T \models P$ but $T \not\models P'$, one can chose nondeterministically a $P_i \in [P]$ and then chose nondeterministically T such that $T \models P_i$ but $T \not\models P'_j$ for every $P'_j \in [P']$.T can be chosen in polynomial time from a set of canonical models as in [17], but $[P']$ is of exponential size. □

Obviously, if $\mathbf{P}'$ has no variables, the results of [17] hold, and containment has a Co-NP solution.

3.2.2 Satisfaction

The next concept needed to define implementations is that of *satisfaction*. The satisfaction relation will be used to compare the shape of trees returned by a function, and the expected result depicted in an interface. Here again, we assume that patterns use variables ranging over finite domains. The satisfaction relation will hence compare return heads of functions, that is patterns defined using only the child relation, and arbitrary patterns, using child and descendant relation. It is defined as follows :

Definition 10 (satisfaction) *Let* $\mathbf{P} = (\mathbb{P}, cond_P)$ *and* $\mathbf{Q} = (\mathbb{Q}, cond_Q)$ *be two patterns possessing one constructor node in each of their trees and such that* $\mathbf{Q}$ *does not involve the relation // (descendant). Let* $[\mathbf{P}]$ *and* $[\mathbf{Q}]$ *be the sets of possible returns of* $\mathbf{P}$ *and* $\mathbf{Q}$*, respectively, see (9). Referring to (10), say that* $\mathbf{Q}$ satisfies $\mathbf{P}$*, written*

$$\mathbf{Q} \sqsubseteq \mathbf{P} \quad \textit{if} \quad \mathbf{Q}_\emptyset \models \mathbf{P}_\emptyset, \textit{ and } \mathbf{Q} \subseteq \mathbf{P}$$

Let us detail this relation. Pattern $\mathbf{Q}$ should be seen as a constraint on the shape and values of forests returned by a service (this is why we can assume that $\mathbf{Q}$ does not use the child relation), and $\mathbf{P}$ as the description of the expected returned values described by an interface. The first property that we want to ensure is that $\mathbf{Q}_\emptyset \models \mathbf{P}_\emptyset$. This is required because when no matching exists, a query nevertheless returns a forest (computed by removing all subtrees below constructor nodes). We feel that is is important to differentiate the cases where a service did not find a matching. Consider for instance a service that may return the empty forest (this is the case when all trees in the return head have constructors as roots) : as this forest satisfies all patterns, then it satisfies the expected results in the interface. By requiring that $\mathbf{Q}_\emptyset \models \mathbf{P}_\emptyset$, we allow interface designers to specify whether services are allowed to return the empty forest or not. Note that as $\mathbf{Q}_\emptyset$ contains no variable, and as $\mathbf{Q}$ uses only child relation between nodes, we can write $\mathbf{Q}_\emptyset \models \mathbf{P}_\emptyset$ without overloading our definitions.

The second property required by satisfaction focuses on forests that are returned by a services when matchings for its return body exist. We want that all forests F returned by a service (i.e. forests F such that $F \models [\mathbf{Q}]$) also satisfy $[\mathbf{P}]$. This means that we want to test whether $[\mathbf{Q}] \subseteq [\mathbf{P}]$ (with $[\mathbf{Q}]$, $[\mathbf{P}]$ seen as disjunctions of patterns). We have seen in theorem 2 that this is equivalent to testing $\left[\mathbf{Q}^1\right] \subseteq \left[\mathbf{P}^1\right]$, and hence $\mathbf{Q} \subseteq \mathbf{P}$.

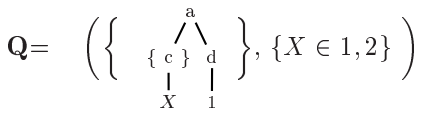
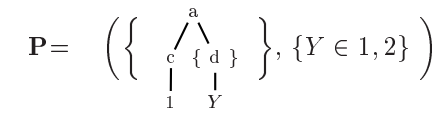
$\mathbf{Q}=$  $\mathbf{P}=$ 

FIG. 10 – Satisfaction relation between patterns

The example of Figure 10 shows two patterns $\mathbf{P}$ and $\mathbf{Q}$ such that $\mathbf{Q} \subseteq \mathbf{P}$ but $\mathbf{Q}_\emptyset \not\models \mathbf{P}_\emptyset$. Note that nothing forces P and Q to contain the same constructors, and that constructor nodes do not play any role is containment satisfaction. However, they are essential to differentiate empty returns.

We can reuse the complexity statement of theorem 4 for satisfaction :

Corollary 1 *Let* $\mathbf{P}$ *and* $\mathbf{Q}$ *be two patterns whose variables range over finite domains, with respectively* l_P *and* l_Q *leaves. Then, testing whether* $\mathbf{Q} \sqsubseteq \mathbf{P}$ *is a Co-NexpTime problem, and can be done in* $O\left((\|\mathbf{Q}\|+1).l_P^{l_Q}.|Q^3|.|P^2|\right)$.

Proof: The complexity statement comes easily from the number of possible valuations in $\left[\mathbf{Q}^1\right]$, $\left[\mathbf{P}^1\right]$, and from the complexity of pattern matching in theorem 1. □

Note that even if theorem 2 avoids testing a doubly exponential set of patterns ($[\mathbf{Q}]$ can contain any subset of valuations of variables of Q, due to constructors), the overall complexity of satisfaction still implies an exponential blowup. The following theorem provides an effective test to check for satisfaction :

First of all, we need to extend the definition of matching to (tree) patterns.

Definition 11 (matching between patterns) *Let* $P = (M_P, G_P, \lambda_M^P, \lambda_G^P)$ *and* $Q = (M_Q, G_Q, \lambda_M^Q, \lambda_G^Q)$ *be two tree patterns. A matching of* P *into* Q *is a a mapping* μ *from the nodes of* P *to the nodes of* Q *such that :*

1. *the root of* P *is mapped to the root of* Q,
2. *the descendant and child relations of the tree pattern are respected by* μ, *that is for every edge* (n,m) *of* P, *if* $\lambda_G^P(n,m) = /$, *then* $\mu(m)$ *is a child of* $\mu(m)$ *in* T, *and if* $\lambda_G^P(n,m) = //$, *then* $\mu(m)$ *is a descendant of* $\mu(n)$ *in* T.

3. *μ preserves the labels, that is for every node n of P such that $\lambda^P_M(n) \in \Sigma \cup \bar{\mathcal{F}} \cup \mathcal{D}$, $\lambda^Q_M(\mu(n)) = \lambda^P_M(n)$. Note that this does not apply to nodes of P labelled by $\star$, which can be mapped by μ to nodes labelled by any symbol or value;*
4. *Nodes of P labeled with variables are mapped to nodes of Q with variables or data values. If a variable node is mapped to a data value, the image of variable nodes must belong to the domain of variables, that is if $\lambda^P_M(n) = X \in \mathcal{V}$ then $\lambda^Q_M(\mu(x)) \in Dom(X)$. If a variable node n of P is mapped onto a variable node of Q, then $Dom(\lambda^Q_M(\mu(n)) \subseteq Dom(\lambda^P_M(n))$. Furthermore, if two nodes are labelled by the same variables, then μ sends them onto nodes of Q with compatible values and/or domains.*

We will say that a tree pattern P embeds *another tree pattern Q, written $Q \models P$ when there is at least one matching from Q to P.*

The definition easily extends to sets of patterns. This definition extends the classical definition of pattern homomorphisms [6, 24, 17] with constraints on variables. It is well known that the existence of homomorphism between patterns is a sufficient condition for pattern containment, but that this is not a necessary condition (see [24] for counter-examples). However, testing the existence of an homomorphism is usually easier than other techniques.

Theorem 5 *Let $\mathbf{P}$ and $\mathbf{Q}$ be two patterns as in definition 10 and possessing disjoint sets of variables. If there exists a matching μ from $\mathbf{P}$ to $\mathbf{Q}$ such that conditions 1 and 2 below hold, then $\mathbf{Q} \sqsubseteq \mathbf{P}$.*

1. *the restriction of μ to nodes in $\mathbf{P}_\emptyset$ is a matching from $\mathbf{P}_\emptyset$ to $\mathbf{Q}_\emptyset$*
2. *The following implication holds : $\left(cond_Q \wedge \left[\bigwedge_{\lambda(n)=X} X = \lambda(\mu(n))\right]\right) \implies cond_P$.*

Proof: Suppose that there exists a matching μ that maps nodes of $\mathbf{P}$ onto nodes of $\mathbf{Q}$, and respects labeling, child and descendant relation, and satisfies condition 1. Then, we obviously have $\mathbf{Q}_\emptyset \models \mathbf{P}_\emptyset$.

Then, there exists a mapping ψ_i from $\mathbf{Q}$ to every F_i belonging to $[\mathbf{Q}^1]$ that maps nodes of $\mathbf{Q}$ to nodes of F_i, and respects labeling and child relation (F_i is the pattern obtained by replacing each variable by one of its valuations). The mapping $\psi_i \circ \mu$ from $\mathbf{P}$ to F_i associates a value to every variable of $\mathbf{P}$, that is it defines a valuation v_i of variables of $\mathbf{P}$. As $Cond_Q \wedge \left(\bigwedge_{\lambda(n)=X} X = \lambda(\mu(n))\right)$ implies $cond_P$, we have that $v_i \in \{\!\{cond_P\}\!\}$. We also have $v_i(\mathbf{P}) \in \left[\mathbf{P}^1\right]$. There exists a bijective morphism ψ'_i from $\mathbf{P}$ to $v_i(\mathbf{P})$ that respects labeling, child and descendant relations and sends nodes of $\mathbf{P}$ onto nodes of $v_i(\mathbf{P})$. It is now easy to see that $\psi \circ \mu \circ \psi_i^{-1}$ is a matching from $v_i(\mathbf{P})$ to F_i, hence $F_i \models \left[\mathbf{P}^1\right]$. As this property holds for every $F_i \in \left[\mathbf{Q}^1\right]$, it also holds for every F such that $F \models \mathbf{Q}$, which concludes the proof. □

The relations between $\mathbf{P}, \mathbf{Q}, v_i(\mathbf{P}), F_i$ are depicted on the diagram of Figure 11.

Theorem 5 provides means to test for satisfaction in $O\left(l_q^{l_p}(C_{cond} + |Q|^3.|P|^2)\right)$, where C_{cond} is the cost for verifying that condition 2 holds, as finding a correct μ can be brought back satisfaction of a pattern. Note however that if the existence of a mapping guarantees satisfaction, the converse is not necessarily true. The reason is that when there are two forests F_i and F_j of $\left[\mathbf{Q}^1\right]$ that satisfy $\left[\mathbf{P}^1\right]$, it means that there exist two patterns $v_i(\mathbf{P})$, $v_j(\mathbf{P})$ that are almost isomorphic up to valuations of leaves, and two corresponding matchings μ_i from $v_i(\mathbf{P})$ to F_i and μ_j from $v_j(\mathbf{P})$ to F_j. F_i and F_j are also quasi-isomorphic up to labeling of leaves. However, distinct

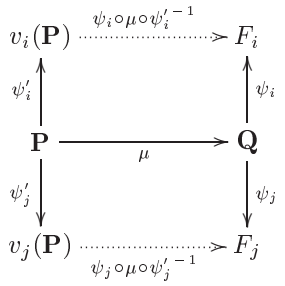


FIG. 11 – Relations between patterns and their valuations

valuations can force μ_i and μ_j to send images of variables nodes in $v_i(\mathbf{P})$, $v_j(\mathbf{P})$ onto nodes of F_i and F_j that are the images of different nodes of $\mathbf{Q}$. We may have $\psi_i^{-1} \circ \mu_i \circ \psi_i' \neq \psi_j^{-1} \circ \mu_j \circ \psi_j'$. Hence, an unique morphism μ may not exist, even when $\mathbf{Q} \sqsubseteq \mathbf{P}$.

Consider for instance the two patterns $\mathbf{P}$ and $\mathbf{Q}$ of Figure 12, where variables X, Y, Z, T, U take value in domain $\{1, 2, 3, 4, 5, 6, 7\}$, variables X', Y' take values in domain $\{3, 4, 5\}$ and with conditions $Cond_Q = [X < Y < Z < T < U]$ and $cond_P = [X' < Y']$. Clearly, there is no unique matching μ from $\mathbf{P}$ to $\mathbf{Q}$ such that for every valuation v_i satisfying $cond_Q$, $\mu(v_i)$ satisfies $cond_P$. Intuitively, for each v_i, variables X' and Y' can always be sent onto nodes of $v_i(\mathbf{Q})$ while meeting condition $cond_P$, but they have to be mapped onto different nodes of $\mathbf{Q}$ for every valuation v_i. This counter example shows that conditions listed in theorem 5 are only sufficient conditions. Note that even when patterns do not contain variables, this test is only a sufficient condition, as it relies on the homomorphism technique for pattern containment [17].

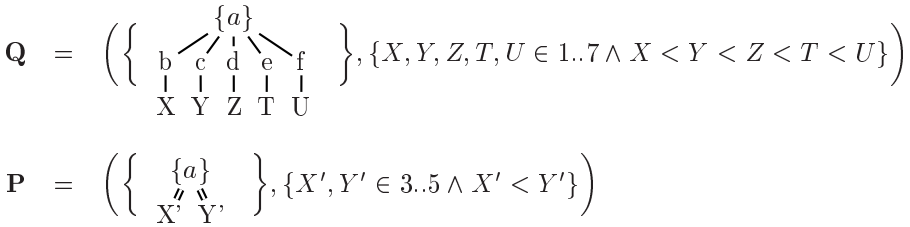


FIG. 12 – A counter example to show that conditions of theorem 5 are only sufficient

3.2.3 Implementation of an interface

We are now ready to define the concept of implementation, that guarantees compatibility between an interface expressing some needs, and a function providing a service :

Definition 12 (implementation) *Let* $\left(f, G_f^c, Q_f^c, \left\{(G_f^r[k], Q_f^r[k])\right\}_{k \in K_f}\right)$ *be a function, where*

$$\begin{aligned} Q_f^c &= Body^c \rightarrow Head^c \\ \forall k \in K_f,\quad Q_f^r[k] &= Body^r[k] \rightarrow Head^r[k] \end{aligned}$$

and let $\left(I, \mathbf{P}_I^{\mathrm{C}}, \{\mathbf{P}_I^{\mathrm{R}}[j]\}_{j\in K_I}\right)$ *be an interface. We say that* f implements I*, written* $f \Vdash I$ *if and only if the following holds :*

$$\underbrace{Body^c \supseteq \mathbf{P}_I^{\mathrm{C}}}_{\text{containment}} \quad \text{and} \quad \underbrace{\forall k \in K_f, \exists j \in K_I : Head^r[k] \sqsubseteq \mathbf{P}_I^{\mathrm{R}}[j]}_{\text{satisfaction}} \tag{12}$$

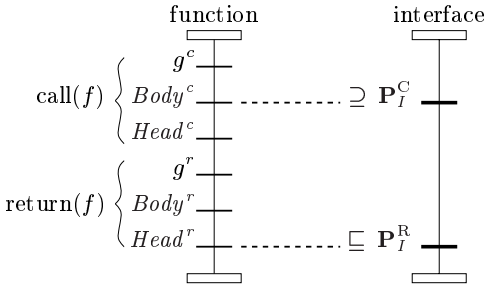


FIG. 13 – Relations between an interface and a function implementing it

Discussion. Let us discuss our design choices for this notion. First, when relating implementation f to interface I, the intent is that the two are located on *different* peers, say q and p. The key remark is that p *does not have direct access to documents located at* q but can only query them by performing a distant call. In particular, p cannot evaluate whether a call guard holds in a document sitting at q. Consequently, our implementation relation does not involve the call guard of f.

Second, in SOA, the notion of *compatibility* relates the required and provided service interfaces and does not involve detailed implementations. To stick to this usual framework, just consider that the *service provided* by f is the pair

$$\left(Body^c, \{Hear^r[k]\}_{k\in K_f}\right)$$

Then, our notion of implementation as in definition 12 is exactly the notion of compatibility of interfaces in SOA.

Recall that the call and return patterns of an interface possess disjoint variables sets. We will see in the is an essential assumption to keep implementation decidable. Indeed, allowing a common variable X in $\mathbf{P}_I^{\mathrm{C}}$ and $\mathbf{P}_I^{\mathrm{R}}[j]$ for some j, gives the possibility to encode a reachability problem with an interface that comports only one return pattern : f implements I iff for a given input pattern containing X it returns some pattern containing the same X. In this context of disjoint variables, checking for implementation reduces to two occurrences of a query containment problem.

Theorem 6 *Let* $\left(I, \mathbf{P}_I^{\mathrm{C}}, \{\mathbf{P}_I^{\mathrm{R}}[j]\}_{j\in K_I}\right)$ *be an interface with variables ranging over finite domains. Let* $(f, G_f^c, Q_f^c, \{(G_f^r[k], Q_f^r[k])\}_{k\in K_f})$ *be a function such that the variables in the head of each* $G_f^r[k]$ *range over finite domains. Then deciding whether* $f \Vdash I$ *is a* Co-NexpTime *problem.*

Proof: The theorem is a direct consequence of theorem 4. □

3.2.4 Discussion

The implementation relation of definition 12 is sound, but not complete (it may consider as incorrect implementation that always return values satisfying one of the

return patterns of an interface). The reason for this is that satisfaction holds only when

$$F \models \bigvee_{k \in K_f} [Head_f^r[k]] \Longrightarrow F \models \bigvee_{k \in K_I} [\mathbf{P}_I^{\mathrm{R}}[k']]$$

However, some values in $\bigvee_{k \in K_f} [Head_f^r[k]]$ may not be effectively produced by a return query (independently of the calling context), or may not be produced by a return query when the service was called using parameters satisfying $\mathbf{P}_I^{\mathrm{C}}$. Hence, the set of possible returned values used in the satisfaction relation is only a superset of the set of returned values that can effectively be produced by an service in a given context. Figure 14 illustrates this situation. Let $\left(I, \mathbf{P}_I^{\mathrm{C}}, \{\mathbf{P}_I^{\mathrm{R}}\}\right)$ be an interface,

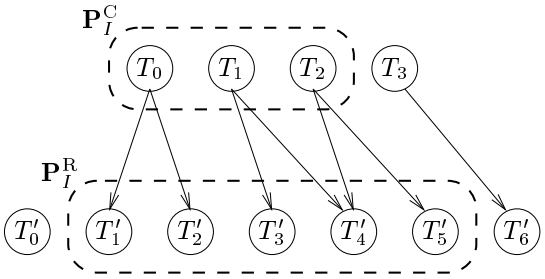


FIG. 14 – Implementation

$(f, G_f^c, Q_f^c, \{(G_f^r, Q_f^r)\})$ be a function (for simplicity, both have singletons as return patterns sets), $T_0, \ldots, T_4$ be a finite set of trees that satisfy $Body^c$ and $T'_0, \ldots, T'_6$ be a set of output trees that can be returned by Q_f^r. In the figure, an arrow from a T_i to a T'_j indicates that T'_j is a possible output for input tree T_i in the context where f is evaluated. The dashed ellipse containing T_i's indicates the trees that satisfy $\mathbf{P}_I^{\mathrm{C}}$, and the dashed ellipse containing T'_i's indicates the trees that satisfy $\left[\mathbf{P}_I^{\mathrm{R}}\right]$. By definition, function f does not implement the interface I since T'_0 and T'_6 do not satisfy $\left[\mathbf{P}_I^{\mathrm{R}}\right]$. On the other hand, we see here that T'_6 does not belong to $f(T_0, \ldots, T_2)$ and that no input to f can produce T'_0. Hence, $f(F) \models \mathbf{P}_I^{\mathrm{R}}$ for any input forest F that satisfies $\mathbf{P}_I^{\mathrm{C}}$. Thus our implementation relation is *sound* (no service that returns a value F such that $\forall k \in K_I, F \not\models \left[\mathbf{P}_I^{\mathrm{R}}[k]\right]$ is accepted as implementation of an interface $\left(I, \mathbf{P}_I^{\mathrm{C}}, \{\mathbf{P}_I^{\mathrm{R}}[k]\}_{k \in K_I}\right)$ but not *complete*. Note however there is no effective solution in general to check whether T'_6 is or is not in $f(T_0, \ldots, T_2)$, because there is no effective procedure to compute the relation between T_is and T'_is induced by f, even if theses sets are finite (services can simulate two counter machines, this point will be detailed in section 4).

This incompleteness may seem bothering, but we will prove in section 4 that determining whether a given value can be returned by a service during the evolution of an AXML system is undecidable, even when variables range over finite domains. Hence, this approximation of returned values is essential, and the implementation relation in definition 12 can be considered as a sound and effective compromise.

Let us illustrate the incompleteness of implementation on an example. Consider for instance the interface I and the service f in Figure 15. It is clear that if service f receives only data such that $X < 10$, then it can only return values with $Y < 10$. However, this dynamic aspect is not captured by our definition of implementation, and service f is not considered as a correct implementation of interface I.

Let us recall that in definition 12, variable sets in $\mathbf{P}_I^{\mathrm{C}}$ and $\{\mathbf{P}_I^{\mathrm{R}}[j]\}_{j \in J}$ are disjoint. The first reason is mainly consistency between variables use and properties that are

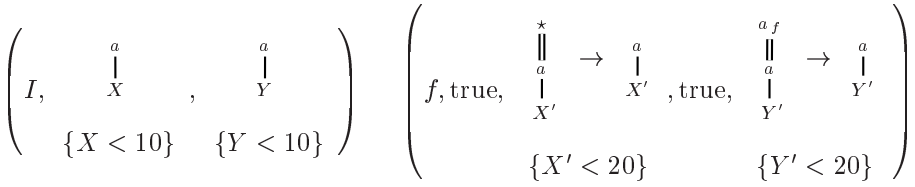


FIG. 15 – Approximation in Implementation relation ; an interface (left) and a function not implementing it (right).

checked for implementation : If we allow $\mathbf{P}_I^{\mathrm{C}}$ and $\mathbf{P}_I^{\mathrm{R}}$ to contain common variables, then condition (12) does not guarantee by itself that common variables have the same valuation in the input and in the output of a service for every correct input document, as relation between $\mathbf{P}_I^{\mathrm{C}}$ and the input to a service and between the output of the service and $\mathbf{P}_I^{\mathrm{R}}$ are checked separately. Hence, the natural semantics is to consider that call and return patterns of an interface are defined over disjoint variable sets.

Changing the semantics of implementation and interfaces to require that common variables designate the same value in $\mathbf{P}_I^{\mathrm{C}}$ and $\mathbf{P}_I^{\mathrm{R}}$ leads to undecidability of implementation. Within this setting, common variables mean that a service that implements an interface should guarantee that some input values are returned by the service. However, checking that the value of a variable is preserved by a service amounts to establishing a relation between input and output ; as already mentioned, this is not effectively computable, even with variables ranging over finite domains. Computing the relation between inputs and outputs resumes to finding whether some output is reachable from an input, which is undecidable, as shown in section 4.1.

3.3 DAXML Schemas and Instances

Now that we have defined services, interfaces, and an implementation relation for interfaces and functions, we are ready to define a compositional Service Oriented Architecture based on AXML through the notion of Distributed AXML *schema*. Active documents over this architecture will be captured by *instances*.

3.3.1 Schemas and their composition

The notion of DAXMLschema captures the architecture of a distributed system, i.e. it collects information on peers of the modeled systems, internal and external services provided or required and their localization, and associates some external services with their actual implementation.

Definition 13 (DAXML schema) *A* Distributed AXML schema *(DAXML) is a tuple*

$$\mathcal{S} = (\mathcal{P}, \Phi_{\mathrm{int}}, \Phi_{\mathrm{ext}}, \mathcal{L}, \gamma)$$

where :

- $\mathcal{P}$ *is a finite set of* peers*, generically denoted by the symbols* p, q *;*
- $\Phi_{\mathrm{int}} \subseteq \mathcal{F}_{\mathrm{int}}$ *is a set of* internal services*, consisting of functions ;*
- $\Phi_{\mathrm{ext}} \subseteq \mathcal{F}_{\mathrm{ext}}$ *is a set of* external services*, consisting of interfaces ;*
- $\mathcal{L} : \Phi_{\mathrm{int}} \cup \Phi_{\mathrm{ext}} \mapsto \mathcal{P}$ *is a function that localizes each internal and external service on a single peer.*

– $\gamma : \Phi_{\text{ext}} \mapsto \Phi_{\text{int}}$ *is a partial function, called* implementation map, *that maps external services to internal services. We furthermore require that* $\mathcal{L}(\gamma(g)) \neq \mathcal{L}(g)$ *and that* $\gamma(g)$ *implements* g ($\gamma(g) \Vdash g$) *for every external service* g. *We do not require that every external service in the system is implemented by an internal service ; when this occurs, the schema is* closed.

In their work [3], Abiteboul, Segoufin and Vianu leave external functions that are not implemented by another peer totally unspecified, with the exception of static constraints expressed as DTD and a boolean combination of patterns. Here, external services are not necessarily unspecified, and define the requirements for a service provided by a distant peer. The intuition behind this definition is that the mapping γ associates an interface definition with the internal service that implements it. The interface defines at the same time the expected output of a service call, but also the valid range of parameters used in a call. When external services are not mapped to the definition of an internal service, their interpretation is the same as in [3], that is they are implemented by other peers that are not part of the system. Note however that even in this case, external services provide precise information that can be used for analysis, as detailed later in this section.

The next step is to equip DAXML with a notion of composition. The intent is that when two schemas compose, one schema provides the details of some external services to the other schema and conversely. To formalize this we first need to define what "providing the details" means. This notion is captured by the notion of pairing map, that associates external service of a given peer to a compatible internal service provided by another peer.

Definition 14 (pairing map) *Consider, for* $i = 1, 2$, *two schemas* $\mathcal{S}_i = (\mathcal{P}_i, \Phi^i_{\text{int}}, \Phi^i_{\text{ext}}, \mathcal{L}^i, \gamma^i)$. *A* pairing map *for* $\mathcal{S}_1$, $\mathcal{S}_2$ *is a partial map*

$$\xi : \left(\Phi^1_{\text{ext}} \cup \Phi^2_{\text{ext}}\right) \setminus \left(D(\gamma^1) \cup D(\gamma^2)\right) \mapsto \Phi^1_{\text{int}} \cup \Phi^2_{\text{int}}$$

where $D(\gamma^i)$ *denotes the domain of* γ^i,*and such that :*

– *For every service* f *in* $D(\xi)$, $\xi(f) \Vdash f$, *i.e.,* $\xi(f)$ *is an implementation of external service* f.
– *For every service* f *in* $D(\xi) \cap \Phi^i_{\text{ext}}$, $\xi(f) \notin \Phi^i_{\text{int}}$, *that is* ξ *maps external services of* $\mathcal{S}_1$ *having no implementation to internal services of* $\mathcal{S}_2$ *and conversely.*
– *For every external service* g, *we have* $\mathcal{L}(\xi(g)) \neq \mathcal{L}(g)$, *that is internal services and the interfaces they implement are located on distinct peers.*

This technique of pairing maps seems adequate to define the composition of DAXML schemas. Indeed, several internal services may be correct implementations of an interface, and relying on names to map a service to an interface assumes that these names are known in advance by peers.

Definition 15 (composition of DAXML schemas) *Let* $\mathcal{S}_1$ *and* $\mathcal{S}_2$ *be two schemas, and let* ξ *be a pairing map. The* composition of $\mathcal{S}_1$ and $\mathcal{S}_2$ through ξ, *denoted by* $\mathcal{S}_1 \|_\xi \mathcal{S}_2$, *is the schema* $\mathcal{S} = (\mathcal{P}, \Phi_{\text{int}}, \Phi_{\text{ext}}, \mathcal{L}, \gamma)$, *where :*

$$\begin{array}{lcl@{\qquad}lcl@{\qquad}lcl} \mathcal{P} & = & \mathcal{P}_1 \cup \mathcal{P}_2 & \Phi_{\text{int}} & = & \Phi^1_{\text{int}} \cup \Phi^2_{\text{int}} & \mathcal{L} & = & \mathcal{L}^1 \cup \mathcal{L}^2 \\ & & & \Phi_{\text{ext}} & = & \Phi^1_{\text{ext}} \cup \Phi^2_{\text{ext}} & \gamma & = & \gamma^1 \cup \gamma^2 \cup \xi \end{array}$$

It is only defined if the two maps $\mathcal{L}^1$ *and* $\mathcal{L}^2$ *agree on the common part of their respective domains, and similarly for* γ^1 *and* γ^2.

A DAXML schema can be seen as kind of a component in a SOA. Note however that nothing forces to schemas to be defined over disjoint sets of peers, interfaces

and services. Schema composition can hence glue systems that are defined over non disjoint sets of peers and functions. Figure 16 illustrates composition. Schema $S1$ is defined over three peers $P1, P2, P3$, with three internal services $f1, f23, f3$ respectively located on $P1, P2, P3$ and three external services g, g' located on P1, and h located on $P3$. Function γ_1 associates g and h with $f2$. Implementation maps are represented in the figure by a dashed line. Schema $S2$ is defined over three peers $P1, P2, P3$, with three internal services $f'1, f'3, f'4$ respectively located on $P1, P3, P4$ and three external services $g'1, h, h'$ respectively located on $P1, P3, P4$. Function γ_2 associates $g'1$ with $f2$, and is represented by a dotted line on the figure. The composition of $S1$ and $S2$ with the pairing map γ that associates g' with $f'4$ and h' with $f3$ is represented on the right of the figure.

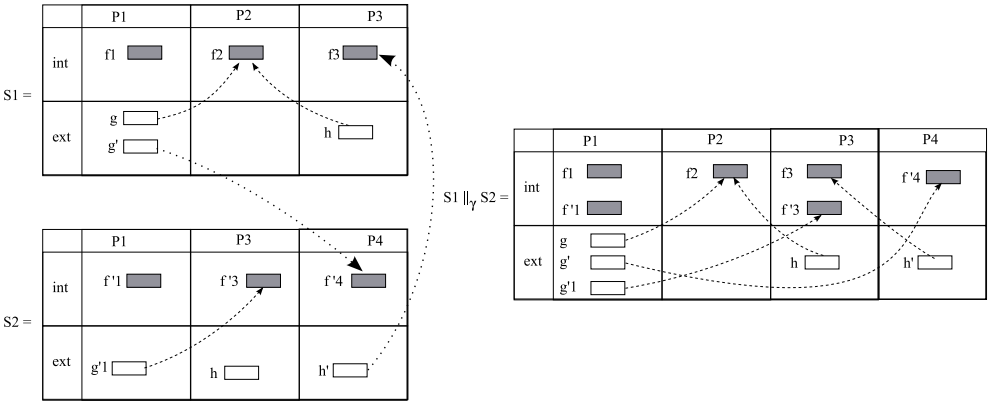


FIG. 16 – DAXML Schemas composition

Theorem 7 *Let $\mathcal{S}, \mathcal{S}_1, \mathcal{S}_2$ be DAXML schemas, then :*

1. *Composition is idempotent : $\mathcal{S} \parallel_\perp \mathcal{S} = \mathcal{S}$ where $\perp$ denotes the trivial pairing map with empty domain.*
2. *Composition is commutative :*

$$\mathcal{S}_1 \parallel_\xi \mathcal{S}_2 = \mathcal{S}_2 \parallel_\xi \mathcal{S}_1$$

3. *If pairing maps $\xi_{12}, \xi_3, \xi_1, \xi_{23}$ satisfy consistency conditions*

$$\forall f \in Dom(\xi_i) \cap Dom(\xi_j) \quad \Rightarrow \quad \xi_i(f) = \xi_j(f), \text{ and} \tag{13}$$
$$Dom(\xi_{12}) \cup Dom(\xi_3) \quad = \quad Dom(\xi_1) \cup Dom(\xi_{23}) \tag{14}$$

where i and j denote two different indices belonging to the alphabet $\{12, 3, 1, 23\}$, then composition is associative :

$$(\mathcal{S}_1 \parallel_{\xi_{12}} \mathcal{S}_2) \parallel_{\xi_3} \mathcal{S}_3 = \mathcal{S}_1 \parallel_{\xi_1} (\mathcal{S}_2 \parallel_{\xi_{23}} \mathcal{S}_3)$$

The proof of these properties of composition are rather straightforward, as composition is defined in terms of unions of sets and functions. Note that condition (13) is trivially met if the sets of external services in $\mathcal{S}_1$ and $\mathcal{S}_2$ are pairwise disjoint.

3.3.2 Instances and their runs

So far moves defined in (8) and (11) describe the evolution of an instance for a schema located on a single peer and considered in isolation. We now describe how

documents located on several peers evolve when these peers provide and use services from one another. The state of such systems is captured by the notion of *DAXML instances*. DAXML instances can be seen as an extension of document instances to sets of peers.

Definition 16 (DAXML instance) *A* DAXML instance *over a schema* $\mathcal{S} = (\mathcal{P}, \Phi_{\text{int}}, \Phi_{\text{ext}}, \mathcal{L}, \gamma)$ *is a tuple* $D = (F, eval, \ell)$*, where* $(F, eval)$ *is a document instance following definition 6, and* $\ell : \text{Trees}(F) \mapsto \mathcal{P}$ *is a function associating a peer to each tree of* F*. We require that, for each node* n *with label* $?f$*, where* $f \in \Phi_{\text{int}}$*, then the following holds :*

$$\ell(eval(n)) = \mathcal{L}(f). \tag{15}$$

Condition (15) expresses that service f is evaluated at the peer that owns it. As for document instances, call and returns of functions make DAXML instances evolve, and we can define runs over DAXML instances. We first focus on moves. Clearly the two internal and external moves, $\vdash_{\text{func}}$ defined in (8), and $\vdash_{\text{intf}}$ defined in (11), extend to DAXML instances respectively for internal services, and external services that have no implementation. Hence, we will have :

- internal call : $(F, eval, \ell) \vdash^{f,\text{call}}_{\text{func}} (F', eval', \ell')$ iff $(F_p = \ell^{-1}(\mathcal{L}(f)), eval_p) \vdash^{f,\text{call}}_{\text{func}} (F'_p, eval'_p)$, where $eval_p$ is the restriction of $eval$ to trees owned by $\mathcal{L}(f) = p$, $F' = (F \setminus \ell^{-1}(\mathcal{L}(f))) \cup F'_p$, and $eval' = eval \cup eval'_p$
- internal return : $(F, eval, \ell) \vdash^{f,\text{ret}}_{\text{func}} (F', eval', \ell')$ iff $(F, eval) \vdash^{f,\text{ret}}_{\text{func}} (F', eval')$, and ℓ' is the restriction of ℓ to trees of F',
- interface call : an interface move an external service I is allowed iff the considered external service is not implemented, that is if $\gamma(I)$ is undefined. If $\gamma(I)$ is defined, we will apply the external call rule define later in the section. $(F, eval, \ell) \vdash^{I,\text{call}}_{\text{intf}} (F', eval, \ell)$ iff $\gamma(I)$ is undefined, and $(F_p = \ell^{-1}(\mathcal{L}(f)), eval_p) \vdash^{I,\text{call}}_{\text{intf}} (F'_p, eval_p)$, $F' = (F \setminus \ell^{-1}(\mathcal{L}(f))) \cup F'_p$,
- interface return : $(F, eval, \ell) \vdash^{I,\text{ret}}_{\text{intf}} (F', eval, \ell)$ iff $(F, eval) \vdash^{I,\text{ret}}_{\text{intf}} (F', eval)$

In addition, the implementation map γ leads to considering another type of move, which consists in calls to services implemented by another peer. Formally, let $(I, \mathbf{P}^{\text{C}}_I, \{\mathbf{P}^{\text{R}}_I[j]\}_{j \in K_I}) \in \mathcal{F}_{\text{ext}}$ be an interface and $(f, G^c_f, Q^c_f, \{(G^r_f[k], Q^r_f[k])\}_{k \in K_f}) \in \mathcal{F}_{\text{int}}$ be a function of a DAXML schema such that $f = \gamma(I)$ (this implies that f implements I). Recall that, in this case, I and f are owned by different peers : $p = \mathcal{L}(I) \neq \mathcal{L}(f) = q$. We will say that there exists a *distant move* from D to D' through the pair $(I, f) \in graph(\gamma)$, written

$$D \vdash^{I,f}_{\text{forw}} D' \tag{16}$$

if one of the following cases holds :

- *Call* : there exists some tree $T \in F$ located of p and containing a node n labeled by $!I$, such that $(T_n, n) \models \mathbf{P}^{\text{C}}_I$, where T_n is the smallest subtree of T containing n and its siblings; Here, T_n represents de parameters of the external call. To avoid finding matchings for $\mathbf{P}^{\text{C}}_I$ into parts of the tree that are not related to current call of I, the matchings are restricted to T_n. F' is obtained by :
  1. changing the label of n to $?I$ and
  2. adding to peer q a tree T' composed of a root labeled by a_I that caps a copy of the smallest forest of T_n containing the matchings $\mu(\mathbf{P}^{\text{C}}_I)$ of $\mathbf{P}^{\text{C}}_I$ into T_n, plus a node labeled by $!f$. Intuitively, the siblings of n can be seen as the parameters of an external call that match an interface requirement.

3. adding to the graph of *eval* the pair (n, T'). Function $eval'$ follows accordingly as an extension of *eval*. Finally, $T' = eval'(n)$ is located in q : $\ell(T') = q$. Observe that, since $(T_n, n) \models \mathbf{P}_I^{\mathrm{C}}$ and f implements I, then $(T', n') \models Body_f^c$, where $Body_f^c$ is the body part of the call query Q_f^c of service f. Note however that nothing forces the guard of service f to hold when the external call is performed, nor after appending tree T' to the document owned by q. The external call is illustrated in figure 17.

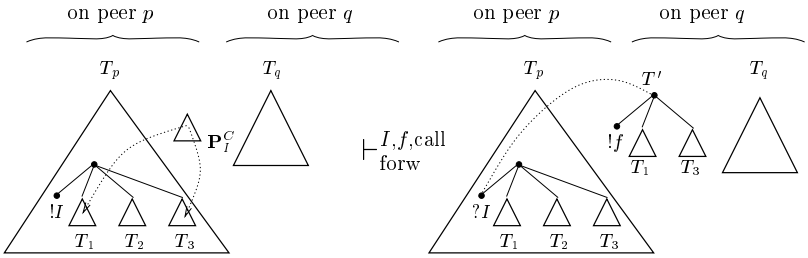


FIG. 17 – An external call

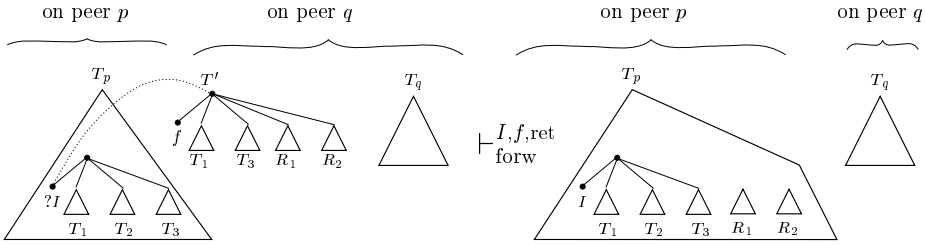


FIG. 18 – Corresponding return

Discussion. Let us discuss our design choices regarding distant calls. Recall that, in accordance with SOA principles, we regard a peer as an autonomous computing unit executing the internal services it owns. Hence, if f sitting at peer q implements interface I sitting at peer p, the only solution for p to make a distant call to f is to communicate the parameters of the call to q. Observe that we only need to communicate the matchings of call pattern $\mathbf{P}_I^{\mathrm{C}}$, thanks to implementation relation of definition 12.

– *Return* : there exists some node n in $T \in F$, labeled by $?I$ and located in p, and some node n' of $eval(n)$ and labeled by f (that is the evaluation of f is completed at its owning peer) ; Then, D' is obtained from D as follows :

1. pick the forest $F_{n'}$ collecting all subtrees that are siblings of n'. This forest may still contain references to interface or service names that are not known by peer p. So, before returning $F_{n'}$, we have to replace some tags of nodes in $F_{n'}$. First, for every node tagged by $!g$, if g is an implementation of an external sevice I owned by p, replace $!g$ by $!I$. If g is not an implementation of an external service of p, leave the node unchanged. The next step is to replace references to external services. For every node tagged by a call to an external service $!J$, if J is implemented by a service g owned by p, replace tag $!J$ by $!g$. If J is implemented by a a service g that is not owned by p, but that implements an interface K of p, replace $!J$ by $!K$. If none of these situation apply, replace $!J$ by a reference $!g$

to the service $g = \gamma(J)$ that implements J. Even after this relabeling, p may receive a forest containing references to services that it does not own. This should not be considered as a problem, as these nodes will simply be ignored by p during next moves.

2. remove $eval(n)$ from F and remove n from the domain of $eval$;
3. change the label on n to I (with no mark). Note that as f implements I, then the forest of appended siblings necessarily satisfies $\left[\mathbf{P}^{\mathrm{R}}_I[j]\right]$ for some $j \in K_I$. The return is illustrated in figure 18.

Note that the return from a distant call can occur as soon as f has been evaluated. The values computed to evaluate f may however still contain references to services calls. All references to external services known by the calling peer are replaced by their interface. The remaining references that are appended are unknown at the calling peer, and can simply be ignored (i.e. they will never be evaluated).

Whenever needed we shall distinguish a call from a return by writing

$$D \vdash^{I,f,\mathrm{call}}_{\mathrm{forw}} D' \quad \text{and} \quad D \vdash^{I,f,\mathrm{ret}}_{\mathrm{forw}} D',$$

respectively. We simply write $D \vdash_{\mathrm{forw}} D'$ to mean $D \vdash^{I,f}_{\mathrm{forw}} D'$ for some pair (I, f) such that $f = \gamma(I)$.

Definition 17 (run) *Let D, D' be DAXML instances. We will say that there is a* move *from D to D', written $D \vdash D'$, if one of the following cases hold :*

- $D \vdash_{\mathrm{func}} D'$,
- $D \vdash_{\mathrm{forw}} D'$,
- $D \vdash^{I}_{\mathrm{intf}} D'$ *for some interface I that has no implementation in the schema.*

We will say that D is deadlocked *if there exists no D' such that $D \vdash D'$. A* run *of a DAXML schema $\mathcal{S} = (\mathcal{P}, \Phi_{\mathrm{int}}, \Phi_{\mathrm{ext}}, \mathcal{L}, \gamma)$ from a document D_0 is an infinite sequence $\rho = D_0, \ldots, D_n, ..$ of instances over $\mathcal{S}$, such that for every i, either $D_i \vdash D_{i+1}$ or D_i is deadlocked and $D_i = D_{i+1}$. We denote by $\mathcal{R}uns(D_0, \mathcal{S})$ the runs of a schema $\mathcal{S}$ starting from instance D_0.*

3.3.3 Projections and restrictions

So far, we have shown how to compose DAXML schemas over sets of peers. We can define the converse operation consisting of a restriction of schemas and instances to a selected subset of its peers. Projections and restrictions will be useful to express and study properties of a subset of a DAXML schema.

Definition 18 (restriction)

1. *Let $\mathcal{S} = (\mathcal{P}, \Phi_{\mathrm{int}}, \Phi_{\mathrm{ext}}, \mathcal{L}, \gamma)$ be a DAXML schema and $\mathcal{R} \subseteq \mathcal{P}$ be a subset of its peers. The* restriction *of $\mathcal{S}$ to $\mathcal{R}$, is the schema*

$$\mathcal{S}_{|\mathcal{R}} = (\mathcal{R}, \Phi_{\mathrm{int}} \cap \mathcal{L}^{-1}(\mathcal{R}), \Phi_{\mathrm{ext}} \cap \mathcal{L}^{-1}(\mathcal{R}), \mathcal{L}_{|\mathcal{R}}, \gamma_{|\mathcal{R}}) \tag{17}$$

where $\mathcal{L}_{|\mathcal{R}}$ is the co-restriction of $\mathcal{L}$ to $\mathcal{R}$, and $\gamma_{|\mathcal{R}}$ is the restriction and corestriction of γ respectively to external and internal function located on peers of $\mathcal{R}$. When $\mathcal{R} = \{p\}$, the restriction is the schema

$$\mathcal{S}_{|p} = (\{p\}, \Phi_{\mathrm{int}} \cap \mathcal{L}^{-1}(p), \Phi_{\mathrm{ext}} \cap \mathcal{L}^{-1}(p), \mathcal{L}_{|p}, \perp) \tag{18}$$

where $\mathcal{L}_{|p}$ is the constant map with value p and $\perp$ is the partial function with empty domain — no interface has an associated implementation.

2. *Similarly, instances $D = (F, eval, \ell)$ over $\mathcal{S}$ can be* restricted *to a subset of peers or to a single peer by setting*

$$D_{|\mathcal{R}} = (\ell^{-1}(\mathcal{R}), eval_{|\mathcal{R}}, \ell_{|\mathcal{R}}) \tag{19}$$

where $eval_{|\mathcal{R}}$ and $\ell_{|\mathcal{R}}$ are respectively the corestriction of *eval* to $\mathcal{R}$ and the restriction and corestriction of ℓ to nodes and trees of $\ell^{-1}(\mathcal{R})$. By the definition of moves (8), (11), and (16), it is clear that moves involve changes in the documents hosted by at most two peers, which can be summarized by the following properties :

- $\forall f \in \Phi_{\text{int}}, \forall \mathcal{R} \subset \mathcal{P}$, $D \vdash^{f}_{\text{func}} D'$ and $\mathcal{L}(f) \notin \mathcal{R}$ implies $D'_{|\mathcal{R}} = D_{|\mathcal{R}}$
- $\forall I \in \Phi_{\text{ext}}, \forall \mathcal{R} \subset \mathcal{P}$, $D \vdash^{I}_{\text{intf}} D'$ and $\mathcal{L}(I) \notin \mathcal{R}$ implies that $D'_{|\mathcal{R}} = D_{|\mathcal{R}}$
- $\forall f \in \Phi_{\text{int}}, \forall I \in \Phi_{\text{ext}} \forall \mathcal{R} \subset \mathcal{P}$, $D \vdash^{I,f}_{\text{forw}} D'$, $\mathcal{L}(I) \notin \mathcal{R}$ and $\mathcal{L}(f) \notin \mathcal{R}$ implies that $D'_{|\mathcal{R}} = D_{|\mathcal{R}}$

Using the above definitions and properties, we are now ready to define projections of runs on a single peer :

Definition 19 (projection of run) *The* projection $\Pi_{\mathcal{R}}(\rho)$ *of a run* $\rho = D_0, D_1, \ldots, D_n, \ldots$ *of a schema* $\mathcal{S} = (\mathcal{P}, \Phi_{\text{int}}, \Phi_{\text{ext}}, \mathcal{L}, \gamma)$ *onto a set of peers* $\mathcal{R} \subset \mathcal{P}$ *is the sequence* $\Pi_{\mathcal{R}}(\rho) = D_{0|\mathcal{R}}, D_{i_1|\mathcal{R}}, D_{i_2|\mathcal{R}}, \ldots D_{i_k|\mathcal{R}}, \ldots$ *such that*

- $0 < i_1 < i_2 < \cdots < i_k \leq \ldots$
- $\forall i_j, j \in 1..k$, *either* $D_{i_j-1|\mathcal{R}}$ *is deadlocked and* $D_{i_j-1|\mathcal{R}} = D_{i_j|\mathcal{R}}$, *or* $D_{i_j-1|\mathcal{R}} \neq D_{i_j|\mathcal{R}}$.
- *For all* n, $0 < i_1 < i_2 < \cdots < i_k \leq n$ *is the largest sequence of indexes satisfying the above condition.*

Intuitively, projections of runs only record the changes of instances that are visible on documents owned by peers in P. Figure 19 shows the projection on peer p for a simple run, involving a local call to function f on peer p, and an external call to a function g offered by peer q where g implements an interface I of p. The projection of this run on peer p hides all trees located on q, and also moves that are local to q, that is evaluation of function g.

Definition 20 (local run) *Let* D_0 *be an instance of a DAXML schema* $\mathcal{S} = (\mathcal{P}, \Phi_{\text{int}}, \Phi_{\text{ext}}, \mathcal{L}, \gamma)$. *A* local run *of* $\mathcal{S}$ *over a subset of peers* $\mathcal{R} \subseteq \mathcal{P}$ *is a run of* $S_{|\mathcal{R}}$ *starting from* $D_{0|\mathcal{R}}$.

Note that it is false in general that $\Pi_{\mathcal{R}}(\mathcal{R}uns(D_0, \mathcal{S})) = \mathcal{R}uns(D_{0|\mathcal{R}}, \mathcal{S}_{|\mathcal{R}})$. There are several reasons for that. The first one ist that when a service implements an interface, it does not necessarily return an answer, as its return guard may never become true. However, local runs may consider a return move of an interface I (hence considering that a service that implements I always returns a value). On the other hand, the set of possible returned values considered in interface moves can be larger than the actual set of values returned by an implementation. The third reason is that local moves do not consider distant calls incoming from peers to which they provide services. When an implementation f located on peer q is called from a peer p, a new tree containing a call to f is created on peer q. Hence, there is a distinction between local changes enforced by external calls (cretation of a tree containing $!f$ for some f) and changes due to a local call to a service (creation of a workspace).

Theorem 8 *Let* $\mathcal{S}$ *be a DAXML schema and* D_0 *be an instance of* $\mathcal{S}$. *Let* $\mathcal{R} \subseteq \mathcal{P}$ *be a set of peers such that* $\mathcal{R}$ *provides no services to* $\mathcal{P} \setminus \mathcal{R}$. *Then,*

$$\Pi_{\mathcal{R}}(\mathcal{R}uns(D_0, \mathcal{S})) \subseteq \mathcal{R}uns(D_{0|\mathcal{R}}, \mathcal{S}_{|\mathcal{R}})$$

Proof sketch: We use the fact that moves are either local, and are then the same in a system or its projection, or distant services calls/returns. Distant calls have the same effects in a system and its projection, and distant returns append values that are compatible with the return pattern of the interface. □

We will see in section 4 that this result is useful to show that some local safety properties of parts of a system are preserved by restriction. Note that when $\mathcal{R}$

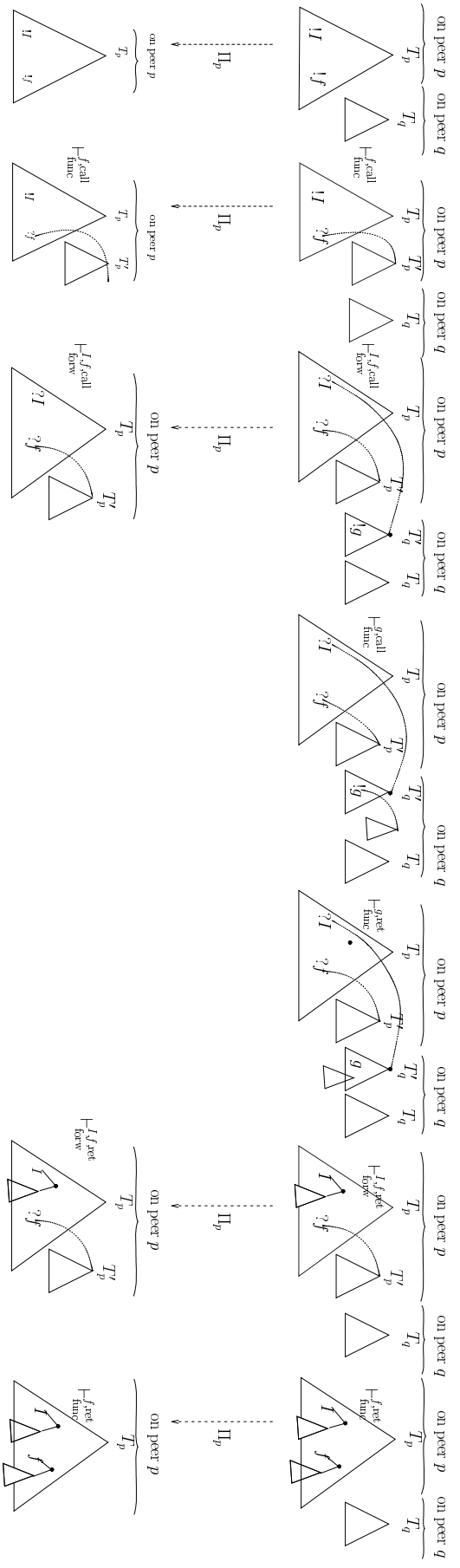


FIG. 19 – Projections of DAXML systems runs

provides services to the rest of the system, then the runs of the restriction of a schema to $\mathcal{R}$ do not take into account the distant calls from peers in $\mathcal{P} \setminus \mathcal{R}$ to services owned by peers in $\mathcal{R}$ that may occur when considering the whole schema. Hence, in this situation, theorem 8 does not hold in general.

3.3.4 Refinement of services and instances

A theory of interfaces should offer comprehensive support for the modular development of systems. One way of achieving this is by offering a notion of *refinement* ensuring substitutability : it should always be possible to replace an interface by a refinement of it and still preserve system properties. As we shall see, our DAXML framework provides interface theories for both services and schemas.

Definition 21 (Service refinement) *Let $\mathcal{S} = (\mathcal{P}, \Phi_{\text{int}}, \Phi_{\text{ext}}, \mathcal{L}, \gamma)$ be a DAXML schema. Say that $\mathcal{S}' = (\mathcal{P}, \Phi'_{\text{int}}, \Phi'_{\text{ext}}, \mathcal{L}', \gamma')$ is a* service refinement *of $\mathcal{S}$ (via external service $I \in \Phi_{\text{ext}}$), written $\mathcal{S}' <_I \mathcal{S}$, if there exists $f \in \mathcal{F}_{\text{int}}$ such that :*

- *f is an implementation of I ;*
- *Φ'_{int} is obtained from Φ_{int} by :*
  1. *adding service f, and*
  2. *renaming, in each service of Φ_{int}, every occurrence of $!I$ by $!f$;*
- *$\Phi'_{\text{ext}} = \Phi_{\text{ext}} \setminus \{I\}$ and γ' is the restriction of γ to Φ'_{ext}.*

Say that a system $\mathcal{S}'$ is a refinement *of a system $\mathcal{S}$ and write $\mathcal{S}' \leq \mathcal{S}$ if and only if there exists a sequence of service refinements $\mathcal{S}' <_{I_k} \mathcal{S}_k \dots \mathcal{S}_1 <_{I_1} \mathcal{S}$.*

Intuitively, service refinement is performed by replacing an interface by an implementation for it. Note that from the definition of refinement, $\leq$ is a partial order (we allow empty sequences of refinements and hence $\mathcal{S} \leq \mathcal{S}$). The notion of refinement also holds for schema instances.

Definition 22 (instance refinement) *Let $D = (F, eval, \ell)$ be an instance over schema $\mathcal{S}$. Say that D' is a* refinement of D via external service $I \in \Phi_{\text{ext}}$ *(written $D' <_I D$) iff D' is obtained from D by relabeling every node labeled by $!I$ by $!f$, nodes labeled by $?I$ by $?f$,and nodes labeled by I by f, where $f \in \Phi_{\text{int}}$ is the name of a service that implements I and is located at the same peer as I. D' is a* refinement *of D if there is a sequence of service refinements $D' <_{I_k} D_k <_{I_{k-1}} \cdots <_{I_1} D$.*

Note that refinement replaces interface moves by function moves. Hence, moves are not necessarily preserved by refinement (up to the renaming of instances).

Theorem 9 *Let D_0 and D_1 be two instances over $\mathcal{S}_0$ and $\mathcal{S}_1$ such that $D_1 \leq D_0$ and $\mathcal{S}_1 \leq \mathcal{S}_0$. Then, for every move $D_1 \vdash D'_1$, there exists two document instances D''_1 and D'_0 such that $D_0 \vdash D'_0$, $D''_1 \leq D'_0$, and $D''_1 \subseteq D'_1$*

Proof: Assume that $D_1 \leq_I D_0$. Then, the move from D_1 to D'_1 can be either a call or a return of a service or interface that is not the refinement of I. In this case, this move is also allowed from D_0, D'_1 is the refinement of the document D'_0 obtained after the move. Now, if the move from D_1 to D'_1 is a call to a service f that refines I, then D'_1 contains a tree labeled with root labeled by a_f (a workspace for an occurrence of a call to f) that is created by the call query of the service. The corresponding interface move allowed from D_0 simply relabels a node with label $!I$ with a label $?I$, without creating new trees. Hence, if we call D''_1 the refinement of I in D'_0, we have that $D''_1 \subseteq D'_1$. Note that the considered move can not be the return of a refined service, as D_0 and D_1 must be isomorphic forests up to renaming of services, and returning from a refined service supposes the existence of a tree created

after a call to f. The reasoning extends to refinement chains of arbitrary length, as illustrated on Figure 20. □

$$\begin{array}{cccccc} D_0 & >_{I_1} & & D_1 & >_{I_2} & & D_2 & \dots \\ \top & & & \top & & & \top & \\ D'_0 & >_{I_1} & D''_1 \subseteq & D'_1 & >_{I_2} & D''_2 \subseteq & D'_2 & \dots \end{array}$$

FIG. 20 – Refinements and runs

Theorem 9 shows the relations between refinement and moves. For a given instance D_0 and its refinement D_1, one could expect a move of the refined system to be a refinement of a move from D_0. However, this is not the case, as instances that appear in moves from D_1 subsume the refinement of moves starting from D_0. The difference between D''_1 and D'_0 sits in the creation of workspaces to evaluate services. This is why runs of refined instances *are not* refinements of runs of less refined schemas. Note also that refinement does not preserve liveness, as an implementation of an interface may never return a value (a call to a service may remain deadlocked, waiting for the satisfaction of its return guard) while interfaces always do. Yet, runs of a schema and runs of its refinements are connected by these subsumption and refinement relation of their reachable instances, and may preserve some safety properties that do not address the contents of the workspaces.

3.4 DAXML versus classical SOA

In this section we provide a summary of comparison between DAXML and SOA principles. Let us compare the services descriptions and composition mechanisms in DAXML with the usual Service Oriented Architectures.

In SOAs, offered services are localized and and made public through *offered interfaces*. Needed services, i.e., the services needed by a peer to provide another service, are specified using *needed interfaces,* usually referred to as *contracts. Service buses* are middlewares playing the role of an adapter between a service and its clients. In addition to this, SOAs can be equipped with a service *repository* referencing the available services, their functionalities, and the way to interact with them.

DAXML systems comprise *functions* that are the counterpart to SOA's services. We did not distinguish services from operations, and considered a single interaction between a client and a service, under the form of an external call. However, guards can be used to define control flows and ensure coordination among external calls. The interfaces owned by a peer are the DAXML counterpart to SOA's contracts. Offered interfaces are not considered in DAXML. It shall, however, be clear that the interface for a service $(f\ ,\ G^c_f\ ,\ Q^c_f\ ,\ \{(G^r_f[k], Q^r_f[k])\}_{k\in K_f})$ is the pair $(Body^r, \{Head^r[k]\}_{k\in K_f})$, where $Body^c$ is the Body part of the call query in service f and $\{Head^r[k]\}_{k\in K_f}$ is the list of all possible Head parts for each return query. Localization of a service is ensured by the pairing map. We did not explicitely define DAXML repositories, but they can be easily defined as the list of services interfaces, with their localizations.

The notion of *compatibility* of a service with a contract is described in DAXML with the notion of implementation. It might be argued that this implementation relation mainly consists in typing separately inputs and outputs to/from a service — whereas one would expect to see a contract as a pair $(In, f(In))$, relating input

parameters to their possible outputs. We will show in section 4 that our design choices for definition 12 of the implementation relation were essential in keeping implementation effective. Still, section 4 explains how to enhance interfaces with *constraints* relating the variables of the call and return queries. These additional constraints, which are not part of implementation relation, are then handled as side entities when performing analysis.

The semantics of implementation and of external call were also designed to take distribution into account. An external call consists in passing a set of parameters to a peer p owning the required service f. Peer p can then use these parameters in combination with data from its own to execute f locally. In particular, the decision to execute f remains local to p and subject to satisfaction of guard G_f^c. Clearly, a consequence of this design choice is that external and internal calls are not similar from the service provider's point of view (note however that this is not a requirement of SOA). But our way is the only way to cope with distribution and at the same time access remote data from a distant database.

Finally, note that the composition of schemas provided in section 3.3.1 is slightly more general than a simple plugin of compatible services. A schema can be seen as a module and composition is not necessarily a parallel composition of schemas over independent peers. Our more flexible notion adds expressiveness at no cost, which proved useful in the Dell example of section 5. We will see in the sequel that the converse operation (i.e. restriction) operation allows compositional reasoning on DAXML schemas.

4 Analysis of DAXML systems

In this section we study some properties of DAXML systems with respect to reachability problems or model checking. Unsurprisingly, many negative results hold, except for the case of *bounded* DAXML systems. Most of the considered properties that we would like to address can be defined as a model checking problem for the Tree-LTL logic proposed by [3] over runs of the system, which is an undecidable problem in the general case.

4.1 Reachability

Reachability is often a key problem for several applications such as verification of safety properties, security,... For an instance D_0 of a schema $\mathcal{S}$, we will say that a pattern P is *reachable* from D_0 if and only if there exists a run $\rho = D_0, \ldots, D_k, \ldots$ in $\mathcal{R}uns(D_0, \mathcal{S})$ such that $D_k \models P$. Such reachability problem is frequently met, when for instance a designer wants to ensure that a bad property, depicted as a patter P_{bad} can not occur. When this simple reachability problem is untractable, then more elaborated questions are usually hopeless. In this section we show that reachability is an undecidable problem for DAXML.

Theorem 10 ([3]) *Let D_0 be an instance of a DAXML schema $\mathcal{S}$, and let P be a tree pattern. Then, it is undecidable whether pattern P is reachable by some run of $\mathcal{S}$ starting from D_0.*

Proof: A DAXML schema can simulate a two counter machine. The encoding of states is as follows : there is a service q_i for each state of the machine, a service c_i for each counter. Being in state q_i correspond to having a running instance of q_i, and the number of running instances of counter c_i encode the value of the corresponding counter. Incrementing and decrementing a counter correspond respectively in launching a new occurrence of c_i or returning from the last called instance of c_i.

The allowed transitions of the machine are then encoded using services guards and intermediate steps to chose the next state to which the machine has to move, perform the needed actions, and set the machine to the desired new state. We will not detail further this encoding, which can be found in [3]. However, it is clear that within this setting, a pattern can encode a desired configuration of the two-counter machine, and hence, reachability of a given pattern is an undecidable problem. □

Note that the two counter machine encoding described above needs only one peer, and does not rely at all on external functions. This encoding is possible as soon as negative patterns in guards and recursive calls between functions are allowed.

Undecidability of reachability has several consequences. First, it is in general undecidable whether a service call terminates, as the return from a call is guarded by a tree pattern, which eventual satisfiability is undecidable. Note also that as it is undecidable whether some pattern is reachable from a given configuration, it is also impossible to compute the image of a set of inputs by a function, as already mentioned in the discussion of section 3.2. This justifies our choice for the incomplete (but decidable) implementation relation.

4.2 SOA modalities

A frequent question addressed in SOA [23, 16], is the modality of a composition. When a component provides a service to another, it makes sense to require that this service eventually returns something (and even in some cases within a bounded amount of time). So far, our implementation definition only specifies that the returned values must satisfy properties described by the return patterns of the interface. However, nothing guarantees that an implementation always returns a value. To complete the description provided by the input and output types that have been introduced in the definition of interfaces 12, one can enhance interfaces with *modalities* "may"or "must". Attaching a *must* modality to an interface I means that a valid implementation f for I must have one of its return guards become eventually true after a call. Giving a *may* modality does not add anything new to our previous definition of implementation (a service may return a forest or not). Note that our may/must modalities slightly differ from the usual modalities proposed for instance in [23, 16], where obligation are attached to transitions of state machines. Here, the obligation imposed by a must modality is not attached to a given move of a DAXML system, but rather forces the occurrence of a return move eventually in the future.

Corollary 2 *Let* $\left(I, \mathbf{P}^{\mathrm{C}}_I, \left\{\mathbf{P}^{\mathrm{R}}_I[k]\right\}_{k\in K_I}\right)$ *be an interface, and let* $f \Vdash I$. *Then it is undecidable whether* f *satisfies a* must *modality.*

Proof sketch: Satisfaction of a must modality can be brought back to a reachability problem (reaching an instance where the return guard of f is satisfied). □

4.3 Beyond static interfaces

The definition of implementation in definition 12 is only a "static implementation", that allows the connection of a service and an interface if every input described in the interface is a valid input of the service and if every output produced by the service is conform to one of the return patterns described in the interface. The input and output patterns of the interface are defined over disjoint sets of variables, nothing forces an implementing service to return a result, and furthermore, when a result is returned, the returned values are not constrained by the parameters of the call. However, in many functions of service description, relating parameters of a call

and values of a return is essential. Imagine, for instance, an application providing best prices to customers, and a service that that takes as input a product name and a maximal price that a customer wishes to pay for it, an returns several offers (i.e pairs (product,price) from a list of product descriptions stored in a database. Clearly, the prices for all proposed items should be lower than the desired input price. However, static interfaces can not force such relation between input and outputs. Now, if we add constraints over variables that appear in the input pattern and in the output pattern of the interface, we can define constraints such as equality of values. This can be seen as defining interfaces which input and output sets of variables may have common elements. This forces us to refine both the notions of interface and implementation to take into account that the expected returned values depend on the values of inputs. We will call this situation *dynamic implementation.*

Definition 23 (dynamic interface) *A* dynamic interface *is a pair* (I, DC)*, where* $\left(I, \mathbf{P}_I^{\mathrm{C}}, \left\{\mathbf{P}_I^{\mathrm{R}}[k]\right\}_{k\in K_I}\right)$ *is an interface, and* DC *is a set of additional constraints over* $\mathcal{X}_I^{\mathrm{C}} \cup \mathcal{X}_I^{\mathrm{R}}$*.*

A function f implements *a dynamic interface (we will use the term* dynamic implementation *and write* $f \Vdash (I, DC)$*) iff* $f \Vdash I$ *and for every pair of forests* (In, Out) *such that* $Out \in f(In)$ *:*

- $In \models \mathbf{P}_I^{\mathrm{C}}$*,*
- $Out \models \left[\mathbf{P}_I^{\mathrm{R}}[k]\right]$ *for some* $k \in K_I$
- *there exists two matchings* $\mu_{In} : \mathbf{P}_I^{\mathrm{C}} \longrightarrow In$ *and* $\mu_{Out} : \mathbf{P}_I^{\mathrm{R}}[k] \longrightarrow Out$ *such that* $v_{\mu_{In}} \cup v_{\mu_{Out}}$ *satisfies* DC*.*

Note that this definition requires the existence of only one single pair of matchings satisfying DC. DC can be seen as an additional set of constraints that restrict the way variable leaves of $\mathbf{P}_I^{\mathrm{C}}$ (respectively $\mathbf{P}_I^{\mathrm{R}}[k]$ are mapped onto leaves of In (respectively O). Hence, matching a dynamic interface means that for every pair input/output provided by f, there is a correct interpretation of variables satisfying the input and output patterns of the interface and also the dynamic constraint.

Corollary 3 *Let* (I, DC) *be a dynamic interface, where* $\left(I, \mathbf{P}_I^{\mathrm{C}}, \left\{\mathbf{P}_I^{\mathrm{R}}[k]\right\}_{k\in K_I}\right)$*, is the static part of the interface, let* $\left(f\ ,\ G_f^c\ ,\ Q_f^c\ ,\ \left\{(G_f^r[k], Q_f^r[k])\right\}_{k\in K_f}\right)$ *be a service, and* $\mathcal{S}$ *be a DAXML schema. Then, it is undecidable whether* $f \Vdash (I, DC)$ *in schema* $\mathcal{S}$*.*

Proof sketch: dynamic implementation can be brought back to a reachability problem. □

Note that our definition of dynamic implementability does not say whether implementability should hold for systems with a given initial schema instance, or for all initial instances of a given schema, or for any schema (that is with arbitrary services, peers,etc.). The undecidability result holds for these three situations.

In the rest of the paper, we will then consider static implementation of interfaces over disjoint sets of variables (over finite domains), and keep aside the verification of *side conditions* involving variables from the input and output patterns.

4.4 Tree-LTL logic

Reachability of a pattern is sometimes not sufficient to express the desired properties of a system, and one has to rely on temporal properties, that depict properties of successive configurations of the system. In this section, we will use the tree LTL logic, as defined in [3], The role of this section is not to study extensively the properties of DAXMLwith respect to tree LTL formulae, but rather to show some limits of verification, and how modularity can help in verification tasks.

Following [3], a Tree-LTL formula is defined by the following grammar :

$$\phi ::= Qpattern \mid \phi \wedge \phi \mid \neg\phi \mid \phi\,\mathcal{U}\,\phi \mid \bigcirc\phi,$$

where *Qpattern* is a Qpattern, i.e., a pattern $\mathbf{P}$ in which variables belonging to some subset $\bar{X}$ are declared as *free* and universally quantified over the formula, and $\wedge, \neg, U, X$ have the usual meaning of LTL. Tree-LTL formulæ are then used to form sentences of the form $\psi = \forall\bar{X}\phi(\bar{X})$, where ϕ is a Tree-LTL formula, and $\bar{X}$ denotes the set of free variables of ϕ. Tree-LTL sentences define properties of runs of DAXML systems. Properties of runs are defined as follows : let $\rho = (D_0, D_1, D_2, \dots)$ be a run of a DAXML system. Then,

- $\rho \models \mathbf{P}$, where $\mathbf{P}$ is a pattern without free variables iff $D_0 \models \mathbf{P}$
- $\rho \models \forall\bar{X}\phi(\bar{X})$ iff for every valuation h of $\bar{X}$, $\rho \models \phi(h(\bar{X}))$
- $\rho \models \neg\phi$ iff $\rho \not\models \phi$
- $\rho \models \phi_1 \wedge \phi_2$ iff $\rho \models \phi_1$ and $\rho \models \phi_2$
- $\rho \models \bigcirc\phi$ iff $\rho' = (D_1, \dots) \models \phi$
- $\rho \models \phi\,\mathcal{U}\,\phi'$ iff $\exists i, (D_i, \dots) \models \phi'$ and $\forall j \leq i, (D_j, \dots) \models \phi$

As usual, properties of runs extends to complete systems with an universal interpretation. We will say that an instance D over a DAXML schema $\mathcal{S}$ satisfies a sentence ψ, written

$$(D, \mathcal{S}) \models \psi$$

if and only of all runs of $\mathcal{S}$ starting from D satisfy ψ. In the sequel, we shall call *system* a pair $(D, \mathcal{S})$ consisting of a DAXML schema and an instance over it.

Existing techniques from [3] allows model checking closed systems where all external services used are effectively implemented — or specified only through data constraints, that are composed of a boolean combination of patterns and of a DTD. A new possibility offered by our framework is to consider *open systems*, that is to use interfaces for services with no implementation when checking for the validity of Tree-LTL sentences. Even if implementations are available, we may still be interested in representing a service by its interface when checking for the validity of Tree-LTL sentences, not by its implementation. The reason is that interfaces are generally more abstract and thus simpler to analyze.

Another interesting question is *local model checking*. Let $(D, \mathcal{S})$ be a system, and $\mathcal{R}$ be a set of peers in $\mathcal{P}$. We can model-check a property locally to $\mathcal{R}$, by considering properties of runs projected on peers in $\mathcal{R}$. We will denote this local model checking problem by

$$(D, \mathcal{S}) \models_{\mathcal{R}} \psi$$

Formally, $(D, \mathcal{S}) \models_{\mathcal{R}} \psi$ holds if there exists a run ρ of $\mathcal{S}$ starting from D, such that $\Pi_{\mathcal{R}}(\rho)$ as defined in definition 19 satisfies ψ.

Abiteboul et al. [3] have shown that without restriction to the considered kind of systems, model-checking of Tree-LTL formulæ is an undecidable problem for guarded AXML. Their results immediately apply to DAXML systems. We will show in this section that, under suitable restrictions, Tree-LTL formulæ can, however, be checked.

Theorem 11 *[3] Let $(D, \mathcal{S})$ be a DAXML system, where $\mathcal{S} = (\mathcal{P}, \Phi_{\text{int}}, \Phi_{\text{ext}}, \mathcal{L}, \gamma)$. Then, the following problems are undecidable :*

1. *for a given Tree-LTL formula ϕ, determine whether $(D, \mathcal{S}) \models \phi$*
2. *for a given Tree-LTL formula ϕ and a set of peers $\mathcal{R} \subseteq \mathcal{P}$, determine whether $S \models_{\mathcal{R}} \phi$*

Proof: These results come directly from [3], and use a reduction from the implication problem for functional and inclusion dependencies [7]. Undecidability of tree LTL holds even for patterns without variables, and for systems composed of a single peer (which immediately brings the undecidability for local model checking). □

Note that the second undecidability result also holds even when $\mathcal{R}$ is composed of a single peer. A possibility to overcome all the undecidability results is to restrict the kind of DAXML systems considered, as proposed in [3]. We now consider AXML trees of *bounded depth*, and set as a new semantics rule that a result is appended to a calling tree if and only if the obtained tree depth does not exceed a constant d (this can be enforced by a very simple DTD). As trees are supposed reduced, this immediately means that the number of bounded trees is limited. This also reduces the different shapes that a tree can have. However, an infinite number of workspaces can still be created by recursive function calls, and an infinite number of different documents can also be produced due to infinity of variable domains.

Definition 24 (Bounded DAXML) *Let* $S = (F, \mathcal{P}, \ell, \Phi_{\text{int}}, \Phi_{\text{ext}}, \mathcal{L}, \gamma)$ *be a DAXML system (under the bounded depth restriction). Let* $\left(f\ ,\ G_f^c\ ,\ Q_f^c\ ,\ \left\{(G_f^r[k], Q_f^r[k])\right\}_{k\in K_f}\right)$ *be one of its internal functions. Let us denote by* $fun(f)$ *the functions* f' *such that there exists* $k \in K_f, Q_f^r[k] = (B_k, H_k)$ *and* H_k *contains a node labelled by* $!f$*. The* call graph *of* S *is a directed graph whose nodes are function names, and whose edges are pairs of function names* (f, f') *such that* $f' \in fun(f)$*.* S *is* bounded *if and only if its call graph is acyclic, and the domain of all its variables is finite.*

Bounded DAXML systems prevent recursion in function calls, hence the number of calls that can be performed from a given document, and consequently the number of workspaces is necessarily bounded.

Theorem 12 *Local and global truth of Tree-LTL formulæ, must modalities of services and dynamic implementation are decidable for bounded systems under the bounded depth restriction.*

Proof sketch: One can easily show that there is only a finite number of documents in a bounded systems under the bounded depth restriction □

Note that considering variables over finite domains only is quite restrictive. The results in [3] shows that for a class of recursion free AXML -without restriction on the variables domains- model checking of tree LTL is decidable. These results might be transferable to our setting. DAXML systems are not always bounded, but by restricting a system to an adequate set of peers of interest, one may bring the model checking problem to a finite or more decidable setting. Now, the question that immediately arises is the relation between the following questions :

i) $(D, \mathcal{S}) \models \psi$

ii) $(D, \mathcal{S}) \models_{\mathcal{R}} \psi$

iii) $(D_{|\mathcal{R}}, \mathcal{S}_{|\mathcal{R}}) \models \psi$

First of all, we can note that as projection results in smaller runs, there is no implication between *i)* and *ii)*? For instance, for formulae containing the next ($\bigcirc$) temporal operator, we can have $(D, \mathcal{S}) \models \bigcirc P$ but $(D, \mathcal{S}) \not\models_{\mathcal{R}} \bigcirc P$, or conversely, even for a single simple tree pattern P without variables. Similarly, $(D_{|\mathcal{R}}, \mathcal{S}_{|\mathcal{R}}) \models \bigcirc P$ means that in all local runs staring from D, P holds in the successor of D. This property is not necessarily preserved by runs of the global system, which may allow more successors to D than local successors (these successors do not need to satisfy P). Conversely, $(D, \mathcal{S}) \models \bigcirc P$ does not imply that $(D_{|\mathcal{R}}, \mathcal{S}_{|\mathcal{R}}) \models \bigcirc P$, as the only

mappings for P might map P onto a workspace that appear in all runs of $(D, \mathcal{S})$ but is ignored in local runs of $(D_{|\mathcal{R}}, \mathcal{S}_{|\mathcal{R}})$. Similar reasoning holds for comparison of $ii)$ and $iii)$ for formulae containing $\bigcirc$, as local runs are a superset of projected runs when $\mathcal{R}$ provides no services to $\mathcal{P} \setminus \mathcal{R}$, and as projections of runs on a set of processes $\mathcal{R} \subseteq \mathcal{P}$ might also contain workspaces created to handling distant calls to services owned by peers in $\mathcal{R}$. Similar reasoning holds in general for formulae of the form $\phi_1 \mathcal{U} \phi_\in$. Hence, we can deduce that projection of runs, restriction, and composition of DAXML schemas an instance do not preserve tree LTL formulae in general.

Note however that if in general $D_{|\mathcal{R}} \models \neg P$ does not imply that $D \models \neg P$, for a simple tree pattern P, $D_{|\mathcal{R}} \models P$ indeed implies that $D \models P$. This leaves the hope that several simple tree LTL properties can be preserved by local runs or projections. In the sequel, we study some useful safety properties of the form "future P" ($FP = (true\,\mathcal{U}\,\mathcal{P})$, "never P", or in the usual terminology $\Box\neg P = \neg(trueUP)$ and "always P", $\Box P = \neg(true\,\mathcal{U}\,\neg\mathcal{P})$.

The following (obvious) lemma shows that projections of runs on a set of peers $\mathcal{R}$ collect all changes to documents that are local to $\mathcal{R}$, and that these documents, and hence the patterns that hold on these documents do not evolve during the moves that are erased by the projection.

Lemma 1 *Let $\rho = D_0, D_1, \ldots D_k, \ldots$ be a run of a DAXML schema $\mathcal{S}$ starting from D_0, P be a tree pattern, and let $D_0 \models_{\mathcal{R}} P$. Let $\Pi_{\mathcal{R}}(\rho) = D_{0|\mathcal{R}}, D_{k|\mathcal{R}}, \ldots$. Then for every $i \in 0..k-1$, $D_i \models P$.*

Proof: All moves from D_0 to D_{k-1} are local evolutions (internal /external calls and returns) of peers in $\mathcal{P} \setminus \mathcal{R}$, as $D_1, \ldots D_{k-1}$ do not appear in $\Pi_{\mathcal{R}}(\rho)$. Hence the matching relation that send every P_i onto trees in $D_{0|\mathcal{R}}$ also exist in D_0, in every $D_{i|\mathcal{R}} = D_{0|\mathcal{R}}$, and in every D_i for $i \in 1..k-1$. □

Theorem 13 *Let D_0 be a DAXML instance, $\mathcal{S}$ be a DAXML schema over a set $\mathcal{P}$ of peers, and $\mathcal{R} \subseteq \mathcal{P}$. Then, the following properties hold :*

i) $(D_0, \mathcal{S}) \models_{\mathcal{R}} \left(\bigwedge_{i\in 1..k} P_i\right)\ \mathcal{U}\ \left(\bigwedge_{j\in 1..k'} P_j\right)$ *implies that*

$$(D_0, \mathcal{S}) \models \left(\bigwedge_{i\in 1..k} P_i\right)\ \mathcal{U}\ \left(\bigwedge_{j\in 1..k'} P_j\right)$$

ii) if $\mathcal{R}$ provides no service to $\mathcal{P} \setminus \mathcal{R}$, then
$(D_{0|\mathcal{R}}, \mathcal{S}_{|\mathcal{R}}) \models_{\mathcal{R}} \Box \bigwedge_{i\in 1..k} \neg P_i$ *implies that* $(D_0, \mathcal{S}) \models_{\mathcal{R}} \Box \bigwedge_{i\in 1..k} \neg P_i$

Proof: Property $i)$ is a direct consequence of lemma 1, as projected runs are sequences of the form $D_{0|\mathcal{R}}, D_{i1|\mathcal{R}}, D_{i2|\mathcal{R}}, \ldots$, obtained by projections of runs of the form $D_0, D_1, \ldots D_{i_1}, \ldots D_{i_2}, \ldots$. Every tree pattern (and hence any conjunction) holding from $D_{i_k|\mathcal{R}}$ to $D_{i'_k|\mathcal{R}}$ also holds in any run obtained by inverse projection from D_{i_k} to $D_{i'_k}$.

Property $ii)$ is a consequence of theorem 8 : if for every local run of $D_{0|\mathcal{R}}, \mathcal{S}_{|\mathcal{R}}$, patterns $P_1, \ldots P_k$ never hold, then they do not hold in $\Pi_{\mathcal{R}}(\mathcal{R}uns(D_0, \mathcal{S}))$, which is a subset of $\mathcal{R}uns(D_{0|\mathcal{R}}, \mathcal{S}_{|\mathcal{R}})$ provided that $\mathcal{R}$ provides no services to $\mathcal{P} \setminus \mathcal{R}$. □

Note that theorem 13 does not mean that the tree LTL properties that are preserved by projection or restriction on a subset of peers are decidable. For instance, $(D_{0|\mathcal{R}}, \mathcal{S}_{|\mathcal{R}}) \models_{\mathcal{R}} \Box\neg P$ can be brought back to the reachability of pattern P.

These preliminary results on local model checking of tree LTL show that there are situations where verification of formulae can be brought back to a finite setting or to smaller sets of runs. Note that projections and restrictions are not automatic solutions, and that interface moves that are used in restrictions may impose exploring more runs than in an implementation. However, we believe that these results are encouraging, and that systematic study of transformation of LTL model-checking into local (or even modular) LTL model checking could bring ad-hoc solutions to reduce the complexity of the problem.

5 A DAXML example : the Dell supply chain

The Dell supply chain is a very interesting example as it combines aspects of Web stored data management — the Dell Web portal — and complex distributed supplier chain involving logistics. The underlying workflow is rich and can be seen as a choreography [18] since there is no central orchestrator. Data are important and complex by involving inventory management. Therefore, locality and abstraction/refinement are key tools in facilitating system analysis. Our study relies on the well documented description [15] and our description below consists of almost verbatim quotes from the above reference.

Verbatim description from [15] : Dell's supply chain works as follows. After a customer places an order, either by phone or through the Internet on www.dell.com, Dell processes the order through financial evaluation (credit checking) and configuration evaluations (checking the feasibility of a specific technical configuration), which takes two to three days, after which it sends the order to one of its manufacturing plants in Austin, Texas. These plants can build, test, and package the product in about eight hours. The general rule for production is first in, first out, and Dell typically plans to ship all orders no later than five days after receipt.

In most cases, Dell has significantly less time to respond to customers than it takes to transport components from its suppliers to its assembly plants. To compensate for long lead times and buffer against demand variability, Dell requires its suppliers to keep inventory on hand in the Austin revolvers (for "revolving" inventory). Revolvers or supplier logistics centers (SLCs) are small warehouses located within a few miles of Dell's assembly plants. Each of the revolvers is shared by several suppliers who pay rents for using their revolver.

Dell does not own the inventory in its revolvers; this inventory is owned by suppliers and charged to Dell indirectly through component pricing. The cost of maintaining inventory in the supply chain is, however, eventually included in the final prices of the computers. Therefore, any reduction in inventory benefits Dell's customers directly by reducing product prices. Low inventories also lead to higher product quality, because Dell detects any quality problems more quickly than it would with high inventories.

Dell has a special vendor-managed-inventory (VMI) arrangement with its suppliers : suppliers decide how much inventory to order and when to order while Dell sets target inventory levels and records suppliers' deviations from the targets. Dell heuristically chose an inventory target of 10 days supply, and it uses a quarterly supplier scorecard to evaluate how well each supplier does in maintaining this target inventory in the revolver. Dell withdraws inventory from the revolvers as needed, on average every two hours. If the commodity is multisourced (that is, parts from different suppliers are completely interchangeable), Dell can withdraw (pull) those components from any subset of the suppliers. Dell often withdraws components from one supplier for a few days before switching to another. Suppliers decide when to send their goods to their revolvers. In practice, most suppliers deliver to their revolvers on average three times a week.

To help suppliers make good ordering decisions, Dell shares its forecasts with them once per month. These forecasts are generated by Dell's line of business (LOB) marketing department. In addition to product-specific trends, they obviously reflect the seasonality in sales. After the center of competence (COC) checks a forecast for predicted availability of components, the forecast goes to Dell's commodity teams and becomes the basis for a six-month rolling forecast that they update weekly. The commodity teams make generic forecasts for systems and components and break those forecasts down to a level of the specific parts that need to be ordered. If the forecast is not feasible, the LOB marketing department revises it, although such

revisions are very rare. The buyer-planner for each commodity receives an updated rolling forecast weekly ; suppliers receive forecasts monthly.

5.1 DAXML modeling of Dell supply chain : architecture and schemas

The overall architecture of the Dell supply chain is shown on Figure 21. As the "Dell supervisor" involves monitoring (a topic in itself) and a lot of algorithmic inventory management, we leave aside this part of the application.

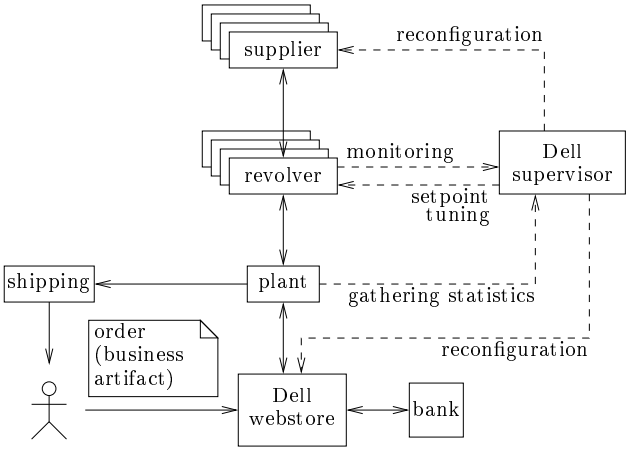


FIG. 21 – Architecture of the example. Multiple boxes indicate that there exist several instances of the considered peer. Each rectangle symbolizes an AXML peer. The customer and adjacent arrows are not within the scope of our model, so we only provide the expectations of the customer in the form of an interface.

The Dell supply chain involves several services, which are summarized in Table 1. Observe that the last three services need access to the revolver.

The distributed and decentralized nature of DSC is reflected through its decomposition into several schemas, in the sense of definition 15. Table 2 lists the schemas with their corresponding services. Corresponding peers are indicated in the caption.

Each column of table 2 specifies a schema. Its internal and external services are indicated by an *int* or an *ext*, respectively. Each schema has a single peer with same name, with the exception of "plant" and "supplier", which share the peer "revolver". Note that we did not make "revolver" a schema — it is, however, a peer. Reason for this is that the revolvers are only passive repositories of items. They exercise no action and offer no service. Access to this repository is given to the two peers "plant" and "supplier". These two peers can both observe and modify the stock at revolver. The revolvers thus serve as kind of "passive shared memory" for these two peers, containing the shared part of their respective documents.

The two services "Deliver" and "AssembleOrder" are owned by the plant (where they are labeled *int*) and used by WebStore via their interfaces (with label *ext*). Accordingly, these two services must be publicized in a registry. Thanks to the fully decentralized execution protocol described in definition 17, we do not need a global registry. Instead, the needed (local) part of the registry is passed as a parameter to the WebStore in the form of the services interfaces when required by the distributed execution.

Also, by specifying as interfaces the services of the WebStore that access the revolver, we will be able to abstract away the interaction between the suppliers and

Service	*Description*
DellOrder	Returns an assembled computer to a customer in conformance with the order form
GiveOrderId	Gives a new identifier to a computer order form
FindOffer	Checks in the DSC system whether there is a special offer (reduced price) for a given model of computer
GivePrice	Returns the actual price of a computer — either catalog price or reduced price in case of offer
CheckCredit	Checks whether a customer has sufficient credit. This service is provided by the bank (which is not part of the DSC, and hence only described as an external service)
RejectOrder	Rejects an order if it can not be processed
ProcessOrder	Processes an order, that is ochestrate the collection of items, their assembling, and the delivery of the assembled computer.
Deliver	Delivers an assembled computer
AssembleOrder	Assembles the items for a computer
GetItem	Gets a given item from the revolver
DecrementInventory	Removes one item from the revolver
RefuelInventory	Adds a certain number of items to the revolver

TAB. 1 – Dell Supply Chain : informal description of the services

schema / *service*	customer	WebStore	plant	supplier
DellOrder	*ext*	*int*		
GiveOrderId		*int*		
FindOffer		*int*		
GivePrice		*int*		
CheckCredit		*ext*		
RejectOrder		*int*		
ProcessOrder		*int*		
Deliver		*ext*	*int*	
AssembleOrder		*ext*	*int*	
GetItem			*int*	
DecrementInventory			*int*	
RefuelInventory				*int*

TAB. 2 – *The Dell system and its schemas.* Schemas Customer and Webstore possess a unique peer with same name. On the other hand, schemas Plant and Supplier possess the peer with same respective name, together with a shared peer Revolver.

the rest of the system. A consequence of this will be that this abstraction will not involve inventory levels (which monitoring is the duty of the suppliers), hence its analysis will not require reasoning about integers. By abstracting some services as interfaces we can thus simplify the task of proving properties — provided that the considered abstraction is precise enough to allow proving the desired property.

5.2 DAXML description of DellOrder

In this description, we will actually use *dynamic interfaces* following definition 23, i.e., interfaces enhanced with additional constraints, also referred to in the sequel as *side conditions*. Side conditions act as assertions when performing program analysis and, therefore, must be discharged at some point.

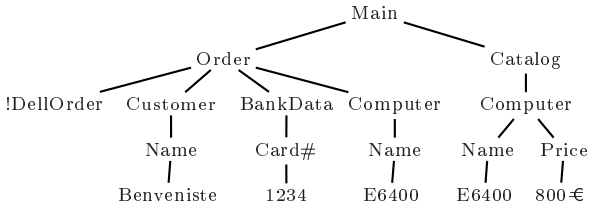


FIG. 22 – *Initial document.* The initial document comprises one order that has been posted to the WebStore. The ordered computer is found in the catalog, together with its price.

An example of initial document is given in figure 22. Then, figures 23–38 describe all internal and external services.

$$\left(\left(I_{\text{DellOrder}}, I^{\text{C}}, I^{\text{R}}\right); [X'_{\text{C}} = X_{\text{C}}] \wedge [X \leq X_{\text{Price}}]\right)$$

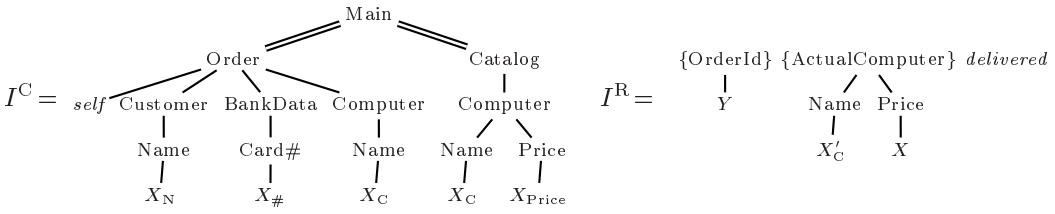


FIG. 23 – *DellOrder, external view from customer.* According to table 2, service "DellOrder" is an external service of the customer and is therefore specified as an interface consisting of two patterns. The meaning of this external service is that when a customer sends a product reference, a catalog price, and his credit card information to the webstore, she expects the delivery of the ordered product at a price that can be lower than the catalog's price if special offers apply to this product, but not greater. Together with its side condition $[X'_{\text{C}} = X_{\text{C}}] \wedge [X \leq X_{\text{Price}}]$, this interface specifies that, when calling DellOrder service, the customer should receive the computer she ordered, at a price not higher than found in the catalog.

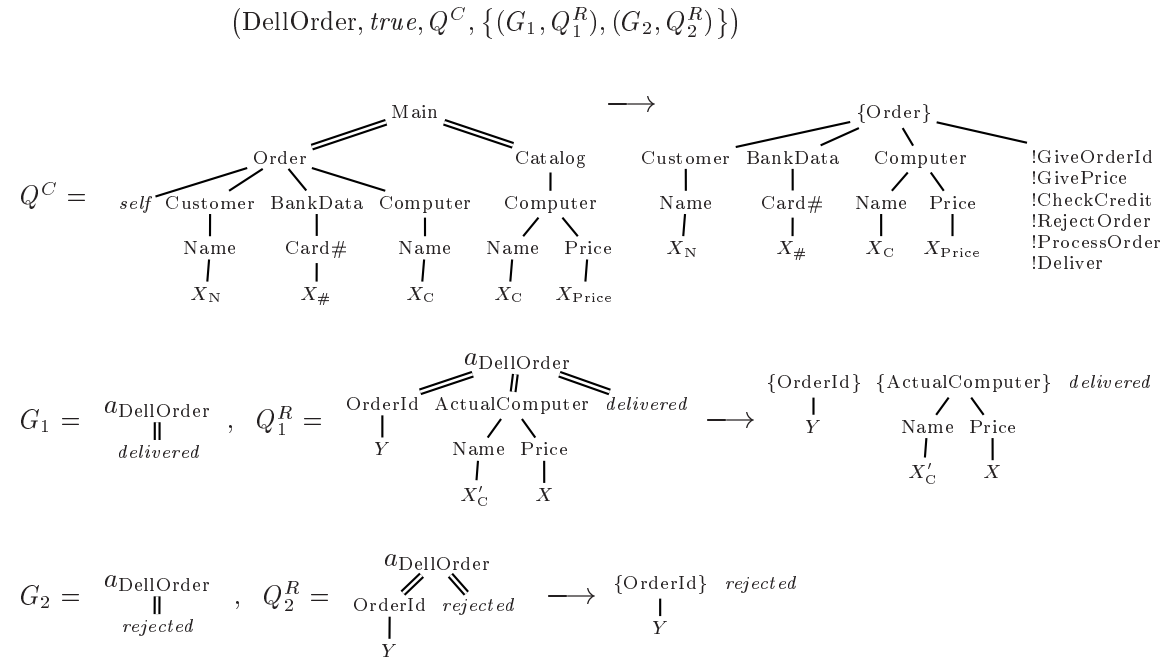


FIG. 24 – *DellOrder, implementation by Webstore.* Let us detail how the Webstore implements the DellOrder. The call guard for the DellOrder service is simply *true*. Note that as this service is only called from a distant site, the value of the guard does not really matter. The call query of the DellOrder copies all the needed parameters and creates instances of service calls that have to be completed before returning an order (this is a flat forest whose nodes are written on top of each other, for better readability). These service calls define the needed steps to build and ship a product. The returned value depends on whether the order was accepted or rejected. Here, the implementation of the DellOrder service cosists of a set of two guarded return queries, namely (G_1, Q_1^R) and (G_2, Q_2^R). Query Q_1^R returns a delivery certificate, but can only be executed when the oder was effectively delivered, which is descibed by guard G_1. Query Q_2^R returns a rejected order when guard G_2 holds, that is when the order was tagged as rejected by service RejectOrder due to negative answer from the bank. Note that these two guards are not specified as being mutually exclusive. Nevertheless, they never hold together at runtime.
The side condition $[X \leq X_{\text{Price}}]$, which is part of the interface of DellOrder (see figure 23), is not discharged by the implementation of DellOrder. It will be, however, discharged by the implementation of GivePrice (see figure 26).
Observe that this is *not* an implementation of the interface specified in figure 23. Reason is that the customer was too optimistic in not considering possible rejection of her payment by the bank. The interface of figure 23 should in fact be corrected accordingly.

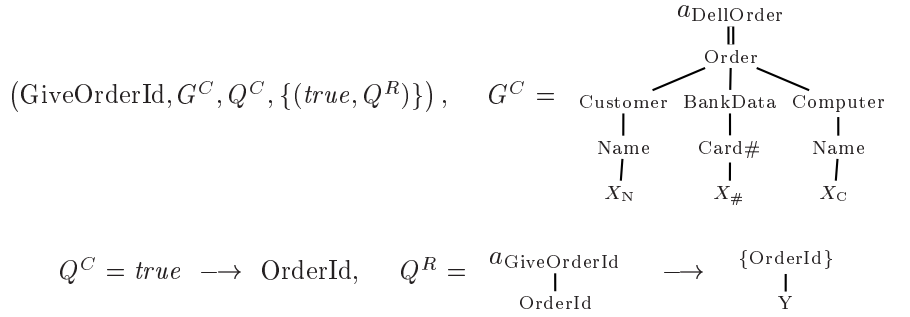


FIG. 25 – *GiveOrderId (internal service of Webstore).* This service is in charge of providing a new identity to an incoming order. The guard of the service G^C checks that the parameters of the order have been correctly filled. The body of the service does not need to identify values in the call, so since guard is true, it can produce immediately a new tag "OrderId". Then, the service has to return a new value that was never used as OrderId. Note that the returned value is represented by a free variable Y, constrained by the fact that Y is a new unused value. Note that GiveOrderId can only be called after a call to DellOrder.

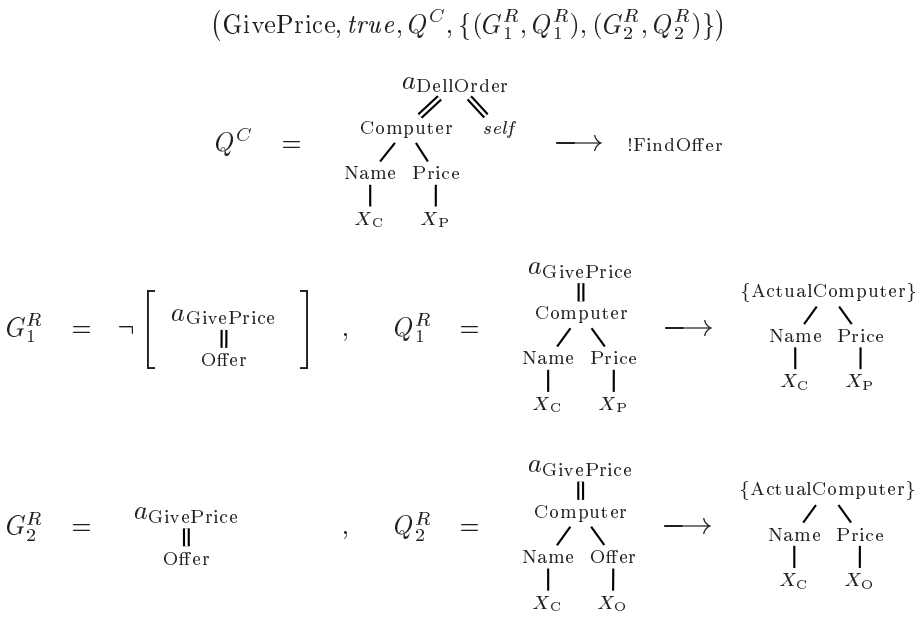


FIG. 26 – *GivePrice (internal service of Webstore).* The role of this service is to find the price of a product in the catalog, and return it. We assume that there is at most one single offer for each product — this can be enforced via a DTD. Two cases can occur, specified by the two guards : either there is an offer on the desired computer, or there is not. This calls for executing service FindOffer before returning a price. We take as an assertion that an offer is always cheaper than the regular price of the catalog : $X_{\text{O}} \leq X_{\text{P}}$.

$$(\text{FindOffer}, true, Q^C, \{(true, Q^R)\})$$

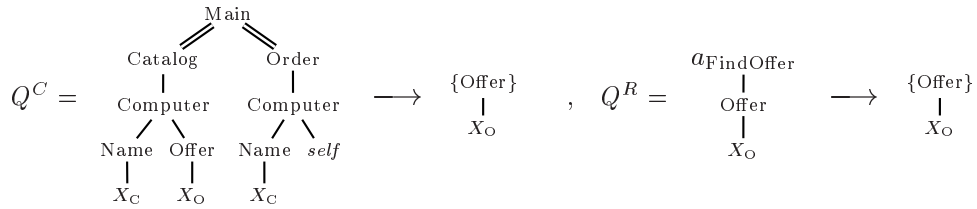


FIG. 27 – *FindOffer (internal service of Webstore).* The role of this service is to find whether an offer exists in the catalog for the requested product. Note that, since a constructor is used in the call and return queries, this service can return an empty forest, indicating that there is no offer on the considered product.

$$(I_{\text{CheckCredit}}, I^{\text{C}}, I^{\text{R}})$$

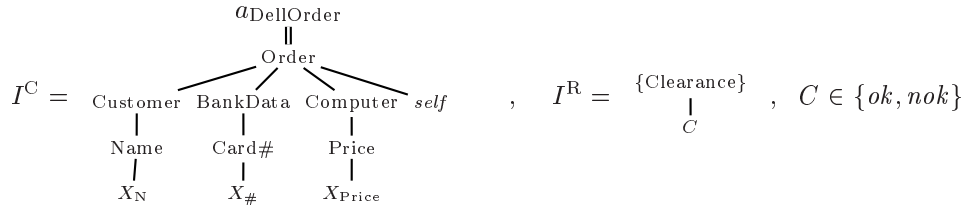


FIG. 28 – *CheckCredit (external view from Webstore).* The webstore uses an external service provided by the bank to verify that the data provided by the customer are correct, and that the bank allows for the payment of the actual price of the product. Note that the presence of tag ActualComputer in the interface definition means in paticular that an external call to CheckCredit can only be performed after the price of the computer has been found by GivePrice. Note also that the shape of I^{C} forces the bank that implements CheckCredit to accept the data the way the webstore structures them. C is a variable taking values in the set $\{ok, nok\}$.

$$(\text{RejectOrder}, G^C, Q^C, \{(true, Q^R)\})$$

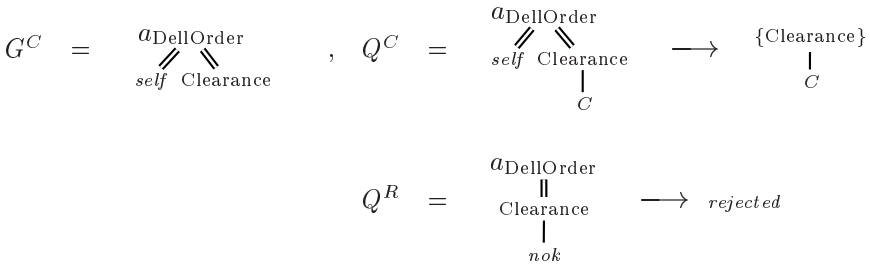
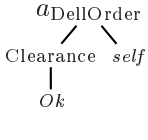
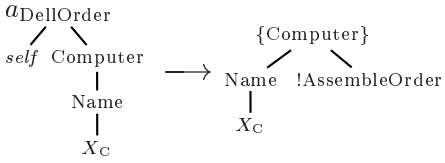
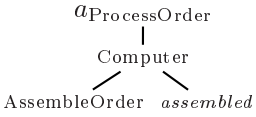


FIG. 29 – *RejectOrder (internal service of Webstore).* This service is in charge of rejecting orders for which the bank did not give clearance. Note that application of this service only results is the appending of tag *rejected* to orders that were not validated by the bank.

$$\left(\text{ProcessOrder}, G^C, Q^C, \{(G^R, Q^R)\}\right)$$

$G^C =$ 

, $Q^C =$ 

$G^R =$ 

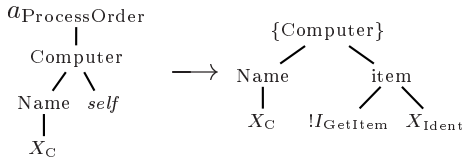
, $Q^R = \textit{true} \longrightarrow \textit{processed}$

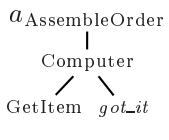
FIG. 30 – *ProcessOrder (internal service of Webstore).* This internal service consists of service that creates an instance of AssembleOrder, and waits for the appearance of a tag "*assembled*" in the workspace before returning a new tag "*processed*". This service cannot be executed before getting *clearance* from the bank, hence, it must be executed after the execution of CheckCredit.

$$\left(I_{\text{AssOrder}}, I^{\text{C}}, I^{\text{R}}\right), I^{\text{C}} = \begin{array}{c}\text{Name}\\ |\\ X_{\text{C}}\end{array}, I^{\text{R}} = \textit{assembled}$$

FIG. 31 – *AssembleOrder (external view from Webstore.)* Service AssembleOrder is performed by the plant, and seen from the webstore as an interface shown in this figure. — In principle, assembling a computer requires getting all its parts, and then assembling them. To simplify the presentation of the example, we have decided that assembling a computer consists in getting the whole computer at once. The more realistic description would not have illustrated any new feature of our modeling. — By looking at this interface, it may seem at a first glance that assembling can be performed as soon as an order is submitted. This is wrong, however, since an occurrence of service AssembleOrder is only created by calling service ProcessOrder, i.e., posterior to obtaining clearance from the bank. Internal service AssembleOrder returns a tag *assembled* when the computer has been assembled. This return can only be performed when the needed parts building the computer have been obtained through service GetItem.

$$\left(\text{AssOrder}, \textit{true}, Q^C, \{(G^R, Q^R)\}\right)$$

$Q^C =$ 

, $G^R =$ 

$$Q^R = \textit{true} \longrightarrow \textit{assembled}$$

FIG. 32 – *AssembleOrder (implementation by Plant).* Calling AssembleOrder creates a new occurrence of GetItem with unique identifier X_{Ident}, to be later used in the management of the stock to ensure that an item is reserved only once. Observe that GetItem is called through its interface. X_{Ident} must be a fresh identifier. The implementation of "AssembleOrder" involves the additional external service GetItem for which we provide an implementation at peer Plant in figure 34.

$$\big(I_{\mathrm{Deliv}}, I^{\mathrm{C}}, I^{\mathrm{R}}\big)\,,\ I^{\mathrm{C}} = \begin{array}{c} a_{\mathrm{DellOrder}} \\ \diagup\ \ |\ \ \diagdown \\ \mathit{self}\ \ \mathrm{Computer}\ \ \mathit{assembled} \\ | \\ \mathrm{Name} \\ | \\ X_{\mathrm{C}} \end{array}\ ,\ I^{\mathrm{R}}_{\mathrm{Deliv}} = \mathit{delivered}$$

$$\big(\mathrm{Deliv}, G^C, Q^C, \{\mathit{true}, Q^R\}\big)\quad ,\quad G^C = \begin{array}{c} a_{\mathrm{DellOrder}} \\ \diagup\ \ \diagdown \\ \mathit{self}\ \ \mathit{assembled} \end{array}$$

$$Q^C = \mathit{true} \longrightarrow \mathit{true}\qquad ,\quad Q^R = \mathit{true} \longrightarrow \mathit{delivered}$$

FIG. 33 – *Deliver (external view from Webstore, top, and implementation by Plant, bottom).* Deliver is an external service for the Webstore (specified as an interface), and is implemented at peer Plant. Both interface (top) and corresponding implementation (bottom) are shown in this figure. Note that Deliver can only be executed when the computer has been *assembled*. The implementation of this service is trivial and returns the tag *delivered*. It is readily checked that the implementation is correct with respect to the given interface.

$$\big(\mathrm{GetItem}, G^C, Q^C, \{\mathit{true}, Q^R)\}\big)$$

$$G^C \quad = \quad \begin{array}{ccc} a_{\mathrm{AssOrder}} & & \mathrm{Main} \\ \| & & \| \\ \mathrm{Item} & & \mathrm{Obtained} \\ \diagup\ \ \diagdown & & | \\ \mathit{self}\ \ X_{\mathrm{Ident}} & & X_{\mathrm{Ident}} \end{array}\ ,\ Q^C = \mathit{true} \longrightarrow \mathit{true},\ Q^R = \mathit{true} \longrightarrow \mathit{got\_it}$$

FIG. 34 – *GetItem (internal service of Plant).* Observe that the guard of GetItem only becomes true after DecInvent has been completed, meaning that an item has been removed from the stock, see the specification of DecInvent.

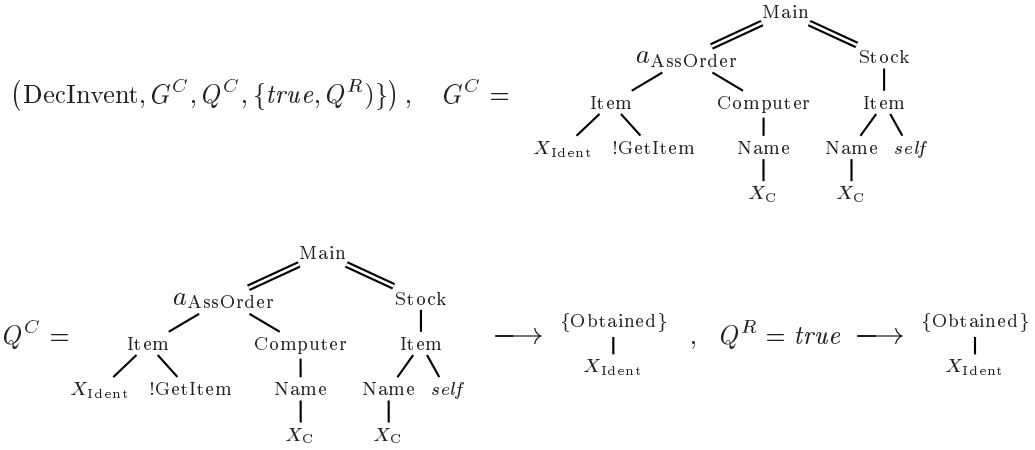


FIG. 35 – *DecrementInventory (implementation by plant).* Occurrences of service DecInvent are attached to each item in the stock at peer Revolver. A "current" status of the stock is depicted in figure 36. Executing DecInvent is interpreted as the removal of one item from the stock. This can only be performed when an occurrence of GetItem is ready for being called. A key observation is that the guard of DecInvent guarantees that the right type of computer is removed from the stock, by identifying the name of Item and the name of Computer with the unique variable X_{C}. On the other hand, this same value X_{C} is consistently copied throughout the processing of the order. *This discharges assertion* X'_{C}=X_{C} *in figure 23.*

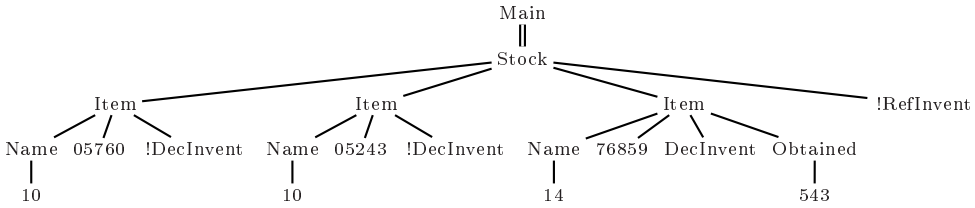


FIG. 36 – *A "current" status of the stock, which is located at peer Revolver.* This figure reads as follows. The first and second subtrees represent items that are available in the stock — this is shown by the presence of!DecInvent, indicating that this item can be removed from stock. The item has a type (here 10) and a unique identifier (here 05760 and 05243, respectively). The last subtree corresponds to an item that has been removed from the stock — this is shown by the removal of "!" before DecInvent. This removal has been given a unique identifier 543. Parts are stocked in the revolver. The stock level is decremented by the plant when getting a part, and incremented by the supplier using service RefuelInventory, see figure 35. Observe that peer Revolver is shared by the two schemas Plant and Supplier.

$$(\text{RefInvent}, G^C, Q^C, \{true, Q^R)\})$$

$$G^C = \neg \left[\begin{array}{c} \text{Main} \\ \| \\ \text{Stock} \\ \text{Item} \quad \text{Item} \quad \text{Item} \\ \text{Name } X \;\; !\text{DecInvent} \quad \text{Name } Y \;\; !\text{DecInvent} \quad \text{Name } Z \;\; !\text{DecInvent} \\ 10 \qquad\qquad 10 \qquad\qquad 10 \end{array} , \; [X \neq Y, Y \neq Z, Z \neq X] \right]$$

$$Q^C = true \longrightarrow true$$

$$Q^R = true \longrightarrow \{\text{Item}\}\ \{\text{Item}\}\ \{\text{Item}\}\ \{\text{Item}\}\ !\text{RefInvent}$$
$$\text{Name } X \;\; !\text{DecInvent} \quad \text{Name } Y \;\; !\text{DecInvent} \quad \text{Name } Z \;\; !\text{DecInvent} \quad \text{Name } T \;\; !\text{DecInvent}$$
$$10 \qquad\qquad 10 \qquad\qquad 10 \qquad\qquad 10$$

FIG. 37 – *RefuelInventory (implementation by supplier).* Service RefuelInventory is an internal service of peer Supplier. This service refuels the stock with a predefined amount of items, when the stock level gets below a predefined critical level. In the implementation specified here, critical level is 3. This is shown by the guard, which expresses that one cannot find three distinct available (indicated by the presence of a !DecInvent) items in the stock. On the other hand, the return query indicates that four new items of type 10 are added to the stock. Finally, the return query regenerates a new occurrence of RefuelInventory. We take as an assertion that all identifiers in the head of the return query are new and pairwise different.

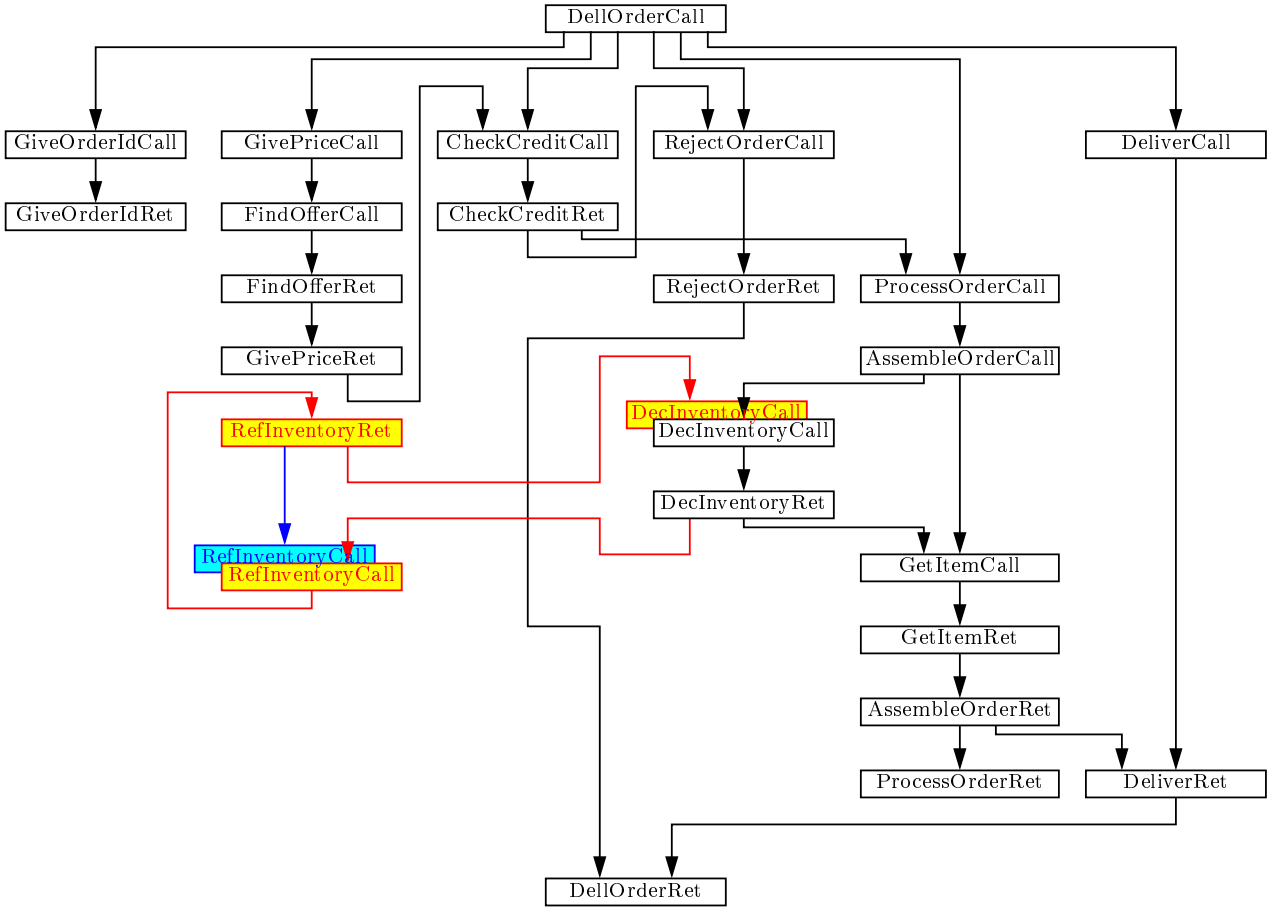


FIG. 38 – *Dependencies between service calls and returns.* This figure shows that several service calls and returns can be performed concurrently. The part of the dependency graph involving the two services "RefuelInventory" and "Decrement Iventory" is slightly involved. It shows, with different colors (namely : black → red → blue), the dynamic instantiation of new occurrences of these services. Observe that no cyclic dependency occurs.

5.3 Using service abstraction to simplify property checking

Service GetItem of figure 34 creates a coupling between the processing of an order and the control of the stock level by the supplier. This intricate inter-dependency makes the analysis of the whole application difficult — and more so if integers are used instead of enumerations while managing stock level.

By using an abstraction in lieu of the above implementation, it is possible to discharge assertion $X'_{\mathrm{C}}{=}X_{\mathrm{C}}$ in figure 23 in a much simpler way. The needed abstraction of implementation of "GetItem" at peer plant, as specified in figure 34, is :

$$(I_{\mathrm{GetItem}}, \mathit{self}, \{\mathit{got\_it}\}) \tag{20}$$

Let us abstract service GetItem as in (20). Since variable X_{C} is copied by all internal services of figures 24–32, we infer that, *if something is returned to the customer, then it must be the right computer*, that is, the one satisfying assertion $X'_{\mathrm{C}}{=}X_{\mathrm{C}}$ in figure 23.

Considering the actual implementation of figure 34 proves, in addition, that 1/ the stock is decremented from the delivered computer, and 2/ delivery eventually occurs (modulo clearance from the bank) since stock gets eventually refueled above critical level provided that refueled amount exceeds critical stock level.

To summarize, our mechanism of abstraction/refinement of services and instances allows to prove some safety properties at a lower cost.

6 Conclusion

Guarded Active XML (GAXML) was proposed by Abiteboul, Segoufin, and Vianu, as a high-level specification language tailored for data-intensive, distributed, dynamic Web services. GAXML consists in XML documents with embedded guarded service calls, thus allowing for the definition of control flows in documents. In this paper we have enhanced GAXML with the concepts needed to satisfy the requirements of "Service Computing" and "Service Oriented Architectures".

We have provided a richer model for external services in the form of *interfaces*. Specifying an interface consists in describing, using patterns : 1/ the shape of documents that can serve as parameters to a call, and 2/ the possible returns of a call. Our notion of interface comes with a notion of *implementation* — a service implements an interface — that builts upon the known concept of containment and a new concept of satisfaction. We have shown that when using variables, this notion of implementation quickly becomes undecidable when the expressive power of interfaces increases. The solution adopted in this paper appears as a good tradeoff between expressiveness and decidability of the implementation relation.

Then, we have proposed *Distributed Active AXML* (DAXML) as a model of guarded active XML systems distributed over a set of peers. Peers transform distributed documents in response to service calls from their own or other peers in an asynchronous way. Our way to capture distribution was by adopting a fine grain semantics for remote service executions. DAXML systems *compose*, thus capturing the mechanism of replacing an external service call by a distant call to an implementation of it, offered by another peer. DAXML systems and documents can be *refined* by replacing, in documents, external service calls by repective implementations thereof; the symmetric operation is service *abstraction*. Abstracting services as interfaces is an efficient tool in simplifying analyses of DAXML documents.

In addition to providing a compositional framework for Web services, the formal semantics of DAXML allows for reasonning on excutions of services using a dedicated logic (in our case Tree-LTL). As for most of expressive languages, this logic

is undecidable, in particular due to undecidability of reachability. Nevertheless, it is still possible to reason about DAXML systems or their abstractions by taking as assertions the eventual satisfaction of return guards. Abstractions are another way to bring back DAXML analysis to model checking of finite systems — for instance, when local safety properties are considered.

We have illustrated our approach on a representative example combining data and workflow management, namely the Dell supply chain. In developing this example, the following features of DAXML proved convenient :

- Allowing shared peers between DAXML schemas was convenient in modeling the *revolver*.
- Side conditions were used to capture the link between ordered and delivered product and its price.
- Checking for implementation relations involved a small number of variables in all cases, thus making the exponential cost of it acceptable.
- The mechanism of refinement/abstraction was useful in two ways : first, it allowed to simplify the proof that the right product would be delivered, if any ; second, it allowed to perform local analyses, from the point of view of a given peer.

Drawbacks and limitations of our approach in handling this example were experienced in the part of the application dealing with stock inventory management. This does not come as a surprise since this kind of application would be naturally handled using arithmetic operations, which were not within our model. This can be overcome by combining the usage of external types and operations together with abstraction mechanisms for reasoning.

Issues of decidability of the implementation relation prevented us from allowing, within interfaces, for side conditions involving call and return variables. Unfortunately, side conditions proved to be unavoidable in practice for nontrivial applications. It seems to us that the way we handled such side conditions was a good compromise.

Another way to cope with undecidability of Tree-LTL (or other temporal logics) would be to concentrate on *business artifacts* [22, 9], i.e., key business relevant objects that flow through a business process specified by a set of services. The analysis of such business artifacts are simpler in that it concerns the analysis of possible evolutions of one item throughout the DAXML system.

Acknowledgements

We would like to thank Serge Abiteboul, Victor Vianu, Luc Segoufin and all our partners of the Docflow project for fruitful discussions on this work.

Références

[1] Serge Abiteboul, Omar Benjelloun, and Tova Milo. The active xml project : an overview. *VLDB J.*, 17(5) :1019–1040, 2008.

[2] Serge Abiteboul, Bogdan Marinoiu, and Pierre Bourhis. Distributed monitoring of peer-to-peer systems. In *ICDE*, pages 1572–1575, 2008.

[3] Serge Abiteboul, Luc Segoufin, and Victor Vianu. Static analysis of active xml systems. In *PODS*, pages 221–230, 2008.

[4] Sihem Amer-Yahia, SungRan Cho, Laks V. S. Lakshmanan, and Divesh Srivastava. Tree pattern query minimization. *VLDB J.*, 11(4) :315–331, 2002.

[5] Henrik Björklund, Wim Martens, and Thomas Schwentick. Optimizing conjunctive queries over trees using schema information. In *MFCS*, pages 132–143, 2008.

[6] Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC*, pages 77–90, 1977.

[7] Ashok K. Chandra and Moshe Y. Vardi. The implication problem for functional and inclusion dependencies is undecidable. *SIAM J. Comput.*, 14(3) :671–677, 1985.

[8] Luca de Alfaro and Thomas A. Henzinger. Interface theories for component-based design. In *EMSOFT*, pages 148–165, 2001.

[9] Alin Deutsch, Richard Hull, Fabio Patrizi, and Victor Vianu. Automatic verification of data-centric business processes. In *ICDT*, pages 252–267, 2009.

[10] Alin Deutsch, Liying Sui, and Victor Vianu. Specification and verification of data-driven web applications. *J. Comput. Syst. Sci.*, 73(3) :442–474, 2007.

[11] Alin Deutsch, Liying Sui, Victor Vianu, and Dayou Zhou. Verification of communicating data-driven web services. In *PODS*, pages 90–99, 2006.

[12] Blaise Genest, Anca Muscholl, Olivier Serre, and Marc Zeitoun. Tree pattern rewriting systems. In *ATVA*, pages 332–346, 2008.

[13] Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient algorithms for processing xpath queries. *ACM Trans. Database Syst.*, 30(2) :444–491, 2005.

[14] Thomas A. Henzinger. Rich interfaces for software modules. In *ECOOP*, pages 517–518, 2004.

[15] Roman Kapuscinski, Rachel Q. Zhang, Paul Carbonneau, Robert Moore, and Bill Reeves. Inventory decisions in Dell's supply chain. *Interfaces*, 34(3) :191–205, 2004.

[16] Kim G. Larsen. Modal specifications. In *Automatic Verification Methods for Finite State Systems*, pages 232–246, 1989.

[17] Gerome Miklau and Dan Suciu. Containment and equivalence for a fragment of xpath. *J. ACM*, 51(1) :2–45, 2004.

[18] Jayadev Misra and William Cook. Computation orchestration. *Software and Systems Modeling*, 6(1) :83–110, March 2007.

[19] Frank Neven and Thomas Schwentick. Xpath containment in the presence of disjunction, dtds, and variables. In *ICDT*, pages 312–326, 2003.

[20] Frank Neven and Thomas Schwentick. Xpath containment in the presence of disjunction, dtds, and variables. In *ICDT*, pages 312–326, 2003.

[21] Frank Neven and Thomas Schwentick. On the complexity of xpath containment in the presence of disjunction, dtds, and variables. *CoRR*, abs/cs/0606065, 2006.

[22] Anil Nigam and Nathan S. Caswell. Business artifacts : An approach to operational specification, journal = IBM Systems Journal. 42(3) :428–445, 2003.

[23] Jean-Baptiste Raclet, Eric Badouel, Albert Benveniste, Benoit Caillaud, and Roberto Passerone. Why are modalities good for interface theories? In *Proc. of the 9th International Conference on Application of Concurrency to System Design (ACSD'09)*, 2009.

[24] Thomas Schwentick. Xpath query containment. *SIGMOD Record*, 33(1) :101–109, 2004.

[25] The Extensible Markup Language (XML) 1.0 (2nd Edition). `http ://www.w3.org/TR/REC-xml`.

A Appendix : proofs

A.1 Proof of Theorem 1

First, let us show that the problem is in NP. It was shown in [13] that evaluation of a pattern without variables (and henceforth existence of a matching) can be done in polynomial time in the size of the tree pattern and of the queried tree (more precisely in $\mathcal{O}(|T|^3 \times |P|^2)$), where $|T|$ and $|P|$ are the sizes of tree T and tree pattern P, respectively. The pattern model proposed in this paper extends the tree patterns considered in [13] with constraints on variables. Note that all valid matchings associate variables in tree pattern P to leaves of tree T. Hence, for a given matching μ, the valuation v_μ must satisfy condition *cond*. Then, finding whether there exists a valid matching can be done in two steps. First, fix the part of the matching μ that associates variables of $\mathbb{P}$ to data leaves of T verify that v_μ must satisfies condition *cond* (this can be done in linear time). The second step consists in verifying that the chosen leaves assignements allows for satisfaction of the rest of the tree pattern, i.e. verifying the satisfaction of a pattern without variables. For this, we can use the polynomial algorithm of [13]. There are at most $|L_T|^{|L_P|}$ such assignments (where L_T and L_P denote respectively the leaves of T and P). Selecting one of them can be done nondeterministically in polynomial time. This also gives us an upper bound in $O\big(|L_T|^{|L_P|} \times (|cond| + |T|^3 \times |P|^2)\big)$ for testing $T \models \mathbf{P}$.

Let us now show the problem is hard. We proceed by reduction from 3SAT, which is known to be NP-complete. Consider a set of variables $V = \{v1, v2, \ldots, v_n\}$ and a logical boolean formula ϕ over V in conjunctive normal form, where each clause contains exactly 3 litterals (i.e. is for instance of the form $(v_{i1} \vee v_{i2} \vee \neg v_{i3})$). We can design a pattern P that has a root, labelled by $\star$, and n children $S_1, \ldots S_n$ such that each node S_i of pattern P is labeled by a variable v_i. Now consider the following tree T, with a root labeled by a tag a, with two children Tt_i and Tf_i, labeled respectively by value $true$ and $false$. We can transform ϕ into an equivalent formula ϕ' as follows : each clause of ϕ is transfomed into an equivalent clause where positive litterals v_i are replaced by an expression of the form $v_i = true$ and every negative litteral $\neg v_i$ by an expression of the form $v_i = false$. Then, it is not difficult to see that $T \models (\{P\}, \phi')$ iff there is a assignment for variables of V that satisfy ϕ. □

A.2 Proof of Theorem 2

We have that $\big[\mathbf{P}^1\big] = \{P_1, \ldots\}$, and for every pattern P in $[\mathbf{P}]$, there exists some P_i in $\big[\mathbf{P}^1\big]$ such that each tree in P_i is a subtree of a tree in P with the same root as in P (P is obtained by replacing constructors by a forest of size greater than 1). Consider a forest F. If $F \not\models P_i$ for some $i \in 1..n$, $F \not\models \mathbf{P}$ for all patterns that contain P_i. Furthermore, if $F \not\models P_i$ for every pattern $P_i \in \big[\mathbf{P}^1\big]$, then $F \not\models [\mathbf{P}]$. If $F \models P_i \in \big[\mathbf{P}^1\big]$, as $P_i \in [\mathbf{P}]$ then $F \models [\mathbf{P}]$. We then have $F \models [\mathbf{P}]$ if and only if $F \models \big[\mathbf{P}^1\big]$. □

A.3 proof of Theorem 3

We reuse a result of [5] showing that containment for conjunctive queries over data trees is undecidable. A conjunctive query is a first order formula that uses unary predicates (the labeling of nodes with σ) and binary predicates of the form $child, child^*, child^+, Nextsibling, Nextsilbling^*, Nextsibling^+$ depicting relation between nodes of an odered tree. [5] also consider data trees, that is trees which nodes carry data from a countably infinite data domain, and add binary predicates $\sim$ and $\neg \sim$ depicting that two nodes carry the same or distinct data value. Bjorklund et

al. show that containment for queries over data trees with $\sim$ and $\neg \sim$ allows for an encoding of PCP, and is hence undecidable. We can reuse their encoding of the PCP, but taking into account two differences between AXML and conjunctive queries :

- in AXML, only leaves can carry data values. This is not really bothering, as we can simulate association of a data by adding to every node a branch that contains a new tag *Data* followed by a data value. Data values are used to encode the choice of indexes in the solution of the PCP.
- in AXML, trees are supposed unordered. The encoding of the PCP with conjunctive queries defines a solution to PCP as a string of the form $\sharp.i_1.w_{i_1}\ldots i_1.w_{i_1}.\sharp.\sharp.i_1.u_{i_1}\ldots i_n.u_{i_n}\sharp$, and forces a counter example to containment to be a string with the predicate $Nextsibling$. However, in the encoding, the ordering between siblings does not matter, and this predicate is only used to capture trees that do no have the shape of a string. Then, the structure of any counter example is of the form depicted in Figure 39. In this figure, r depicts the root of the tree and each w_i must be interpreted as a branch describing word w_i in the given PCP instance. Depicting a counter example that is branching can be imposed by a pattern saying that the minimal distance from the common ancestor of two nodes is greater than 2 (the size of the branch with the *Data* tag). The PCP encoding with Tree patterns uses variables defined over an ordered infinite domain of indexes or over an ordered finite set of letters (the letters used in the considered instance of the PCP), and tags in $\Sigma = r, \sharp, Index, Data$, to identify respectively the root, delimiters of w_i's concatenations, indexes, and the letters of each w_{i_k} or u_{i_k}. Hence, a bad counter example that does not have the form of Figure 39 will necessarily satisfy a pattern of the form depicted in figure 40, where $\sigma_1 \neq \sigma_2 \in \Sigma$.

For the rest of the PCP encoding, we can reuse exactly the same constructs as in [5]. Inequality of values is obtained by using ordered domains, and then imposing constraints of the form $X < Y \vee X > Y$.

□

A.4 Proof of theorem 4

It is known that containment testing for XP(/,//,[ ],⋆,|) (that is for disjunctions of tree patterns) is in Co-NP [19]. This result of [19] reuses a first Co-NP algorithm proposed by Miklau et al. [17] for XP(/,//,[ ],⋆). Let us detail this algorithm.

Let P be a tree pattern and Q be another tree pattern, respectively of size n and m. Then there is a finite set of bounded canonical models $\mathbb{T}(P, sl_Q)$, such that $P \subseteq Q$ if and only if for every $T \in \mathbb{T}(P, sl_Q)$, $T \models Q$. $\mathbb{T}(P, sl_Q)$ is the set of trees that "look like P and are not bigger than$Q$". A tree in the canonical set is a tree built from p, where $\star$ labeled nodes are replaced by a new tag ($\sharp$), and descendant edges by chains of $\sharp$ of size up to $sl_Q + 1$, where sl_Q is the size of the largest chain of nodes labeled by $\star$. This yields a Co-NP algorithm, that consists in choosing in polynomial time a member of $\mathbb{T}(P, sl_Q)$, and then testing (again in polynomial time [13]) if $T \models Q$. If this is not the case, then T is a counter-example showing that $P \not\subseteq Q$. Note that all trees in $\mathbb{T}(P, sl_Q)$ are of size at most in $O(2n.(m+2))$.

The result of [17] has been extended to include disjunctive combinations of tree patterns in [19], just by noticing that $P = P_1 \vee \cdots \vee P_n \not\subseteq Q = Q_1 \vee \cdots \vee Q_l$ if and only if there exists some $i \in 1..n$ such that $T \models p_i$ and $p_i \not\subseteq q$. The algorithm proposed by Neven is to guess a p_i, find a t from $\mathbb{T}(P_i, sl_Q)$ (we can take for sl_Q the largest length of $\star$ chain in all Q_j's), and to check in polynomial time whether $T \models p_i$ and $T \not\models Q$, that is test for every $q_j, j \in 1..l$ that $T \not\models q_j$.

Let us now show how the algorithms of Neven and Miklau adapt to our setting. The tree patterns used in our expressions are exactly the tree patterns of the above algorithms, with some additional constraints on the values of nodes.

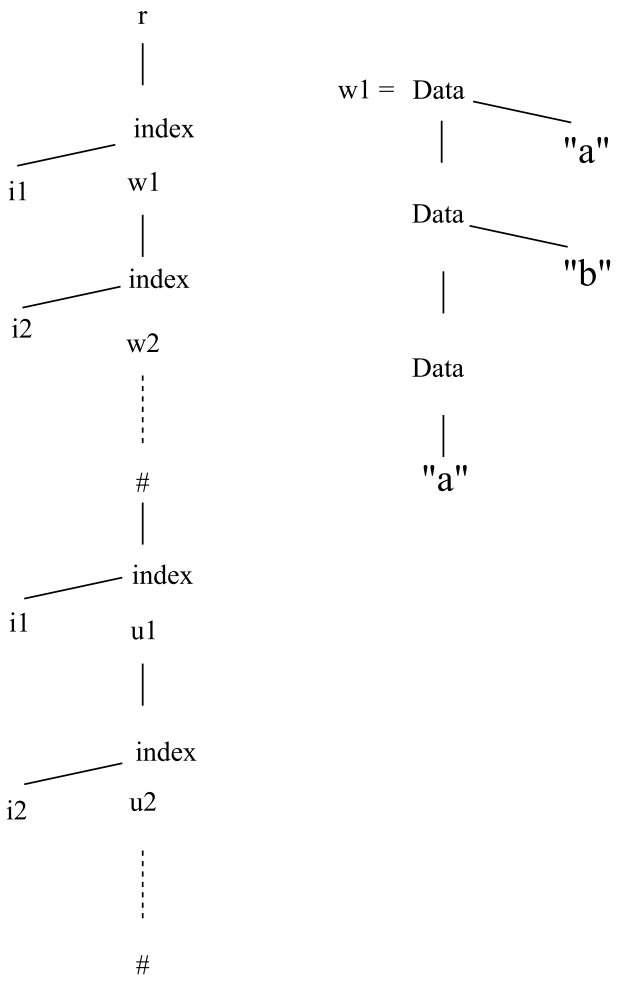


FIG. 39 – Shape of counter examples

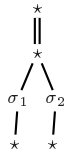


FIG. 40 – Pattern to discriminate trees that are not strings

Let $\mathbf{P} = (\mathbb{P}, cond_P)$ and $\mathbf{Q} = (\mathbb{Q}, cond_Q)$ be two patterns, defined over set of variables $\mathcal{X}_P$ and $\mathcal{X}_Q$. Using the disjunction over patterns defined in (2), consider $\mathbf{P}' = \mathbf{P}_{v_1} \vee \cdots \vee \mathbf{P}_{v_N}$ and $\mathbf{Q}' = \mathbf{Q}_{w_1} \vee \cdots \vee \mathbf{Q}_{w_{N'}}$ where $v_1, \ldots v_N$ and $w_1, \ldots w_{N'}$ are the different possible valuations of $\mathcal{X}_P$ and $\mathcal{X}_Q$ that are compatible with $cond_P$ and $cond_Q$, and $\mathbf{P}_v$, is $\mathbf{P}$ in which all variables have been replaced by their valuation according to v. As all variables are given actual values, the condition attached to $\mathbf{P}_v$ becomes trivial.

There is no containment from $\mathbf{P}$ to $\mathbf{Q}$ if we can find a pattern $\mathbf{P}_{v_i}$ and a tree T such that $T \models \mathbf{P}_{v_i}$ but $T \not\models \mathbf{Q}'$. Note however that the size of these disjunctions can be exponential in the size of variables. That is the algorithm of Neven et al. is not Co-NP anymore, as there is necessarily an exponential number of tests to prove that a T is a counterexample. Then the algorithm to check containment is as follows. Chose in polynomial time a valuation v, verify that it satisfies $cond_P$. If this holds, then choose non deterministically a tree T from the set of canonical models $\mathbb{T}(P_v, sl_Q)$. Here, we can note that this set of canonical models does not depend from any valuation chosen for Q, as sl_Q is the same for every $\mathbf{Q}_{w_i}$. Then, we have to check if $T \models \mathbf{Q}'$, that is test for every possible valuation w_i satisfying $cond_Q$, if $T \models \mathbf{Q}_{w_i}$. If $T \not\models \mathbf{Q}_{w_i}$ for every w_i that satisfies $cond_Q$, then T is a counterexample, and $\mathbf{P} \not\subseteq \mathbf{Q}$ follows. As we have to perform an exponential number of tests to be sure that $T \not\models \mathbf{Q}'$, the containment algorithm is then in Co-Nexptime.

A.5 Proof of theorem 8

Assume that there exists a run $\rho = D_0, D_1, \ldots, D_k, \ldots$ such that $\Pi_{\mathcal{R}}(\rho) \notin \mathcal{R}uns(D_{0|\mathcal{R}}, \mathcal{S})$. Without loss of generality, we can consider that move $D_{k-1} \vdash D_k$ is the incriminated move, that is $\Pi_{\mathcal{R}}(D_0, D_1, \ldots, D_{k-1})$ is a prefix of some run in $\mathcal{R}uns(D_{0|\mathcal{R}}, \mathcal{S})$, but $\Pi_{\mathcal{R}}(D_0, D_1, \ldots, D_k, D_k)$ is not. We know that P provide no services to the rest of the system, $D_{k-1} \vdash D_k$ can then be : an internal call, an internal return, a distant call between $p, p' \in P$, or a return from a distant call, or an interface move (call or return). Internal calls or returns are guarded by patterns evaluated locally to the peer owning the service. Hence, if $\Pi_{\mathcal{R}}(D_{k-1})$ is reachable from $D_{0|\mathcal{R}}$ by a run over $\mathcal{S}_{|\mathcal{R}}$, then $\Pi_{\mathcal{R}}(D_{k-1}) \vdash \Pi_{\mathcal{R}}(D_k)$ is a move of $\mathcal{S}_{|\mathcal{R}}$. An interface call or return in $\mathcal{S}$ is a similar interface call/ return in $\mathcal{S}_{|\mathcal{R}}$. The last possibility is then distant calls or returns. Distant calls are only guarded by the fact that siblings of a node contain a given pattern, defined locally to P by an interface. They are then allowed with similar modifications on trees owned by P by $\mathcal{S}_{|\mathcal{R}}$ and $\mathcal{S}$ (i.e. replacing a tag $!I$ at a given node by a tag $?I$). The last kind of move is a return from a distant call, which consists in appending to the trees owned by P a forest F returned by the invoked service. As the called service implements an interface of P, it returns values that are compatible with the return pattern of the interface, and hence, for every D_k that is reachable from D_{k-1}, we have that $\Pi_{\mathcal{R}}(D_{k-1}) \vdash \Pi_{\mathcal{R}}(D_k)$ is an interface move of $\mathcal{S}_{|\mathcal{R}}$. Assuming $\Pi_{\mathcal{R}}(\rho) \notin \mathcal{R}uns(D_{0|\mathcal{R}}, \mathcal{S})$ then leads to a contradiction. □

A.6 proof of corollary 2

Satisfying a must modality means returning a value, and hence reaching a configuration where a return guard of f holds (one can even restrict to the case where f has only one single guarded return query). However, the call query of service f can create a workspace that can be transformed into an initial configuration of the two counter machine depicted in theorem 10 in a finite number of unavoidable steps (these steps just ensure that the machine is put in a predetermined state q_i, symbolized by a running function $?q_i$). Hence, satisfying a must modality can be

brought back to a halting problem of a two counter machine, and is undecidable.□

A.7 Proof of corollary 3

As for must modality, one can bring back our problem to a reachability problem. Without loss of generality, one can restrict to the case where the interface comports one single return pattern $\mathbf{P}_I^{\mathrm{R}}$, and function f contains only one guarded return query $Q_f^r = (B, H)$. For a pattern $B = (\mathbb{P}, cond_B)$ and a condition $cond$ let us define $B_{\setminus cond} = (\mathbb{P}, cond_B \wedge \neg cond)$ as the a pattern with the additional constraint that $cond$ does not hold.

Suppose that B and H are isomorphic trees, and B and $\mathbf{P}_I^{\mathrm{R}}$) are also isomorphic patterns. Suppose also that there is only one valid valuation of variables in $\mathbf{P}_I^{\mathrm{C}}$ for every input In to function f, that is compatible with $\mathbf{P}_I^{\mathrm{C}}$. Let $cond_I$ be the list of constraints of the form $X = c$ for every value imposed to a variable X of $\mathbf{P}_I^{\mathrm{C}}$ by the input In.

Then, within this setting, f is not a dynamic implementation of I if, for a given input value In allowed by I, there is no reachable workspace where $G_f^r \wedge B_{\setminus cond_I \cup DC}$ holds (that is if f can return a value such that DC is not satisfied). As for must modalities in corollary 2, the call query of f can encode the initial configuration of a two counter machine, and the schema can simulate this machine. Hence, dynamic implementation is undecidable. □

A.8 Proof of theorem 12

Proof: Let S be a bounded DAXML system. Let Σ^S be the set of tags used in a forest or a service of S, and let $\mathcal{D}^S$ be th data values appearing in a forest of S or in a domain of a variable of S. When a tree has maximal d, then the number of non isomorphic subtrees at depth d is $a = |\Sigma \cup \mathcal{F}^! \cup \mathcal{D}^S|$. At depth $d-1$, it is $a.2^a$. Hence, the overall number of reduced trees of depth at most d is $T(a,d) = a.2^{T(a,d-1)}$, with $T(a,0) = a$. The acyclicity of service calls graph guarantees that in any schema derived from S, there is a bounded number of workspace under evaluation. Hence, all Tree-LTL formulæ can be checked over a finite structure as an usual LTL formula, with Patterns holding is a document as atomic propositions. □

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Centre de recherche INRIA Futurs : Parc Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399